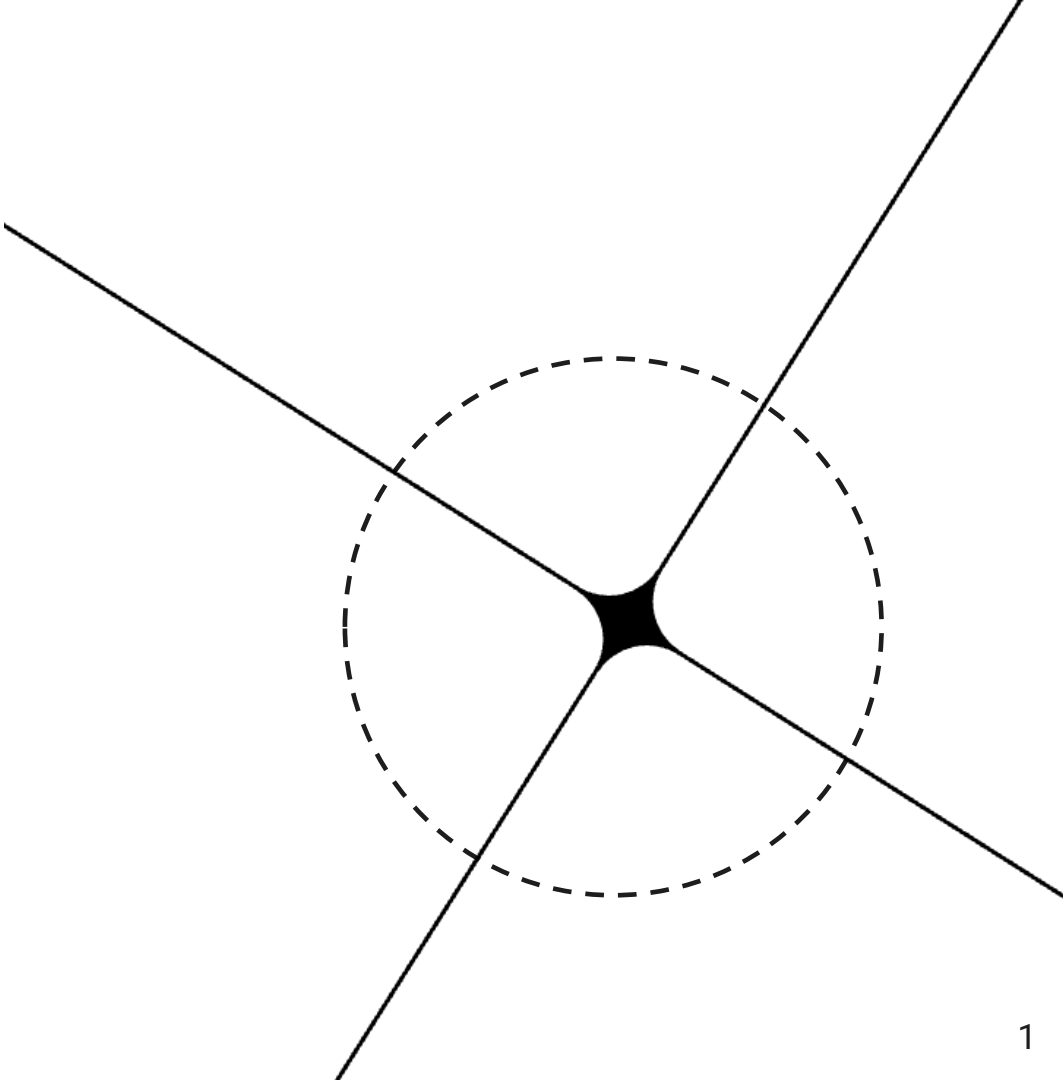
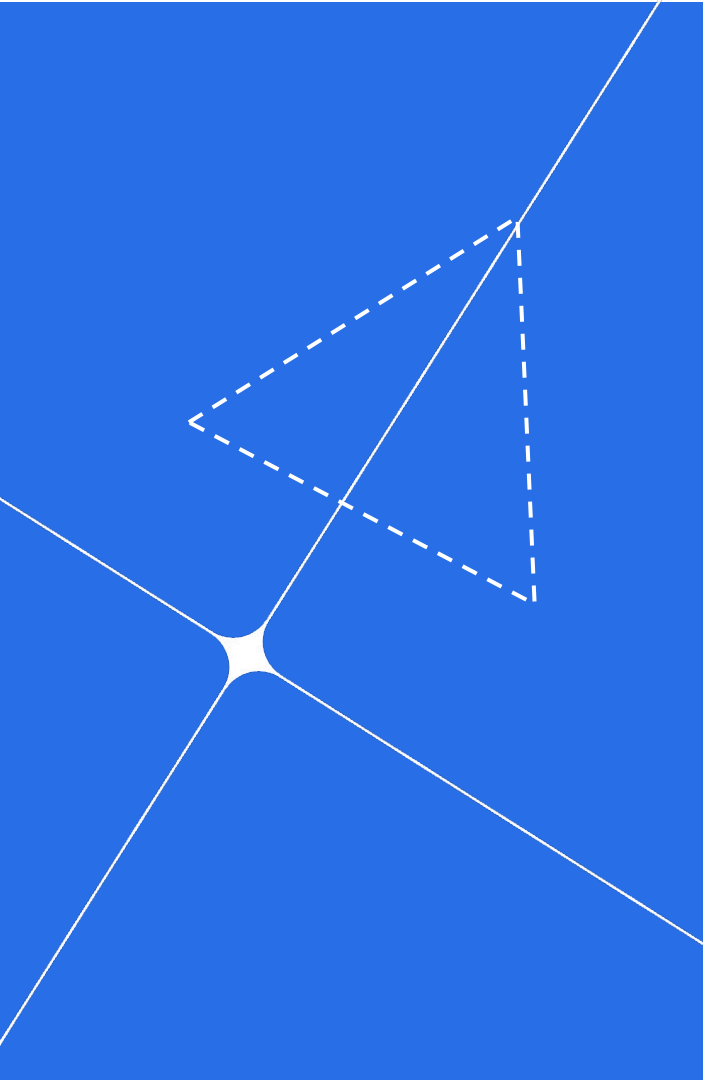
INSTITUT POLYTECHNIQUE DE PARIS

TELECOM SudParis

Génération d'image

Julien Romero

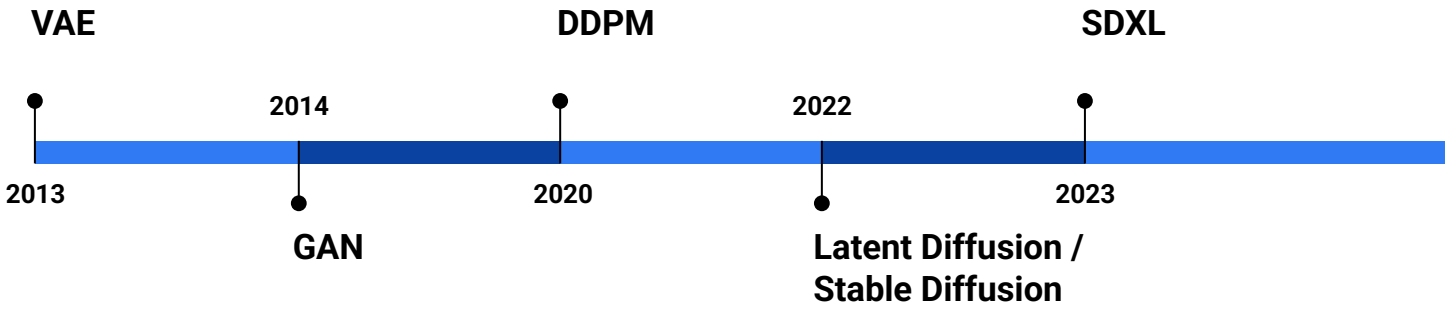




Motivation, panorama et problème de la génération d'images

Pourquoi la génération d'images maintenant ?

- Dans une équipe produit, le goulot = produire des visuels (marketing, UI, catalogues, assets) plus vite que la cadence business
- Génération = **accélérer** la création (variantes), **personnaliser** (par audience/pays), **industrialiser** (templates + prompts)
- Exemples
  - grand public/pro : Adobe Firefly (création/édition d'images intégrée aux workflows créatifs)
  - design à grande échelle : Canva / Magic Media pour produire des images à intégrer directement dans des supports
  - e-commerce : génération/variation de visuels de produits (fonds, déclinaisons) côté Shopify et apps associées
- Même logique en interne entreprise
  - réduire le coût de shooting, accélérer A/B testing créatif, "content ops" plus automatisable
- Problème central du cours
  - **contrôler** la génération (qualité, style, contraintes) tout en restant robuste et reproductible



Qu’est-ce qu’on cherche à optimiser, concrètement ?

- But : produire des échantillons x “crédibles” issus d’une distribution p(x) (ou conditionnée p(x | c))
- 4 axes qui tirent dans des directions différentes : **fidélité**, **diversité**, **contrôlabilité**, **coût/latence**
  - **Fidélité** : textures, détails, absence d’artefacts (mains, texte dans l’image, géométrie)
  - **Diversité** : éviter de “tourner en rond” (pas de *mode collapse*, pas de répétitions visuelles)
  - **Contrôlabilité** : respecter une contrainte (prompt, composition, masque, style, identité visuelle)
  - **Coût** : nombre d’étapes de génération, VRAM, temps GPU, throughput (batching), reproductibilité (seed)
- En pratique : on arbitre selon l’usage (marketing = itération rapide ; imagerie technique = fidélité + contraintes)

Tâches et workflows usuels

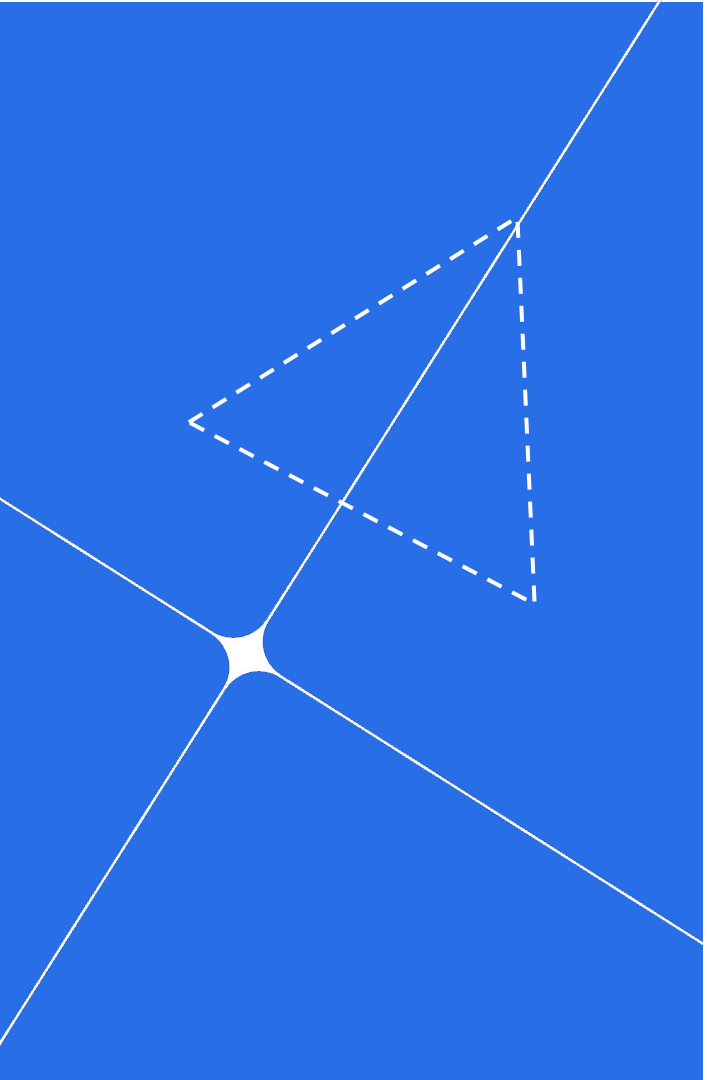
- **Unconditional** : générer “du plausible” sans consigne (utile surtout en recherche/benchmark)
- **Conditional** : générer sous contrainte ccc (texte, classe, image, masque, carte)
  - Text-to-Image : prompt → image (création, moodboards, déclinaisons)
  - Image-to-Image : image source + prompt → variation (style transfer moderne, redesign, “restylisation”)
  - Inpainting / Outpainting : masque → compléter / étendre (suppression d’objet, extension de décor)
  - Super-resolution : faible résolution → haute résolution (restauration, upscale, détails plausibles)
  - Editing “guidé” : conserver structure globale, modifier localement (vêtements, arrière-plan, lumière)

Trois familles de modèles

- VAE : modèle probabiliste latent ; optimise une borne (ELBO)
  - **stable**, bonne couverture, mais détails parfois lissés
- GAN : jeu adversarial G vs D
  - **échantillons très nets**, mais entraînement instable + risque *mode collapse*
- Diffusion / score-based : débruitage progressif
  - **qualité + diversité** + conditionnement efficace, mais sampling itératif
- Lecture “ingénieur”
  - chaque famille propose une stratégie différente pour approximer $p(x)$ ou $p(x \mid c)$
- Choix en pratique
  - qualité/contrôle/compute/latence + contraintes d’intégration (pipeline, sécurité, licence)
- À retenir pour la suite : aujourd’hui, la diffusion domine le text-to-image (ex. SDXL)

De la recherche au produit” : contraintes réelles à garder en tête

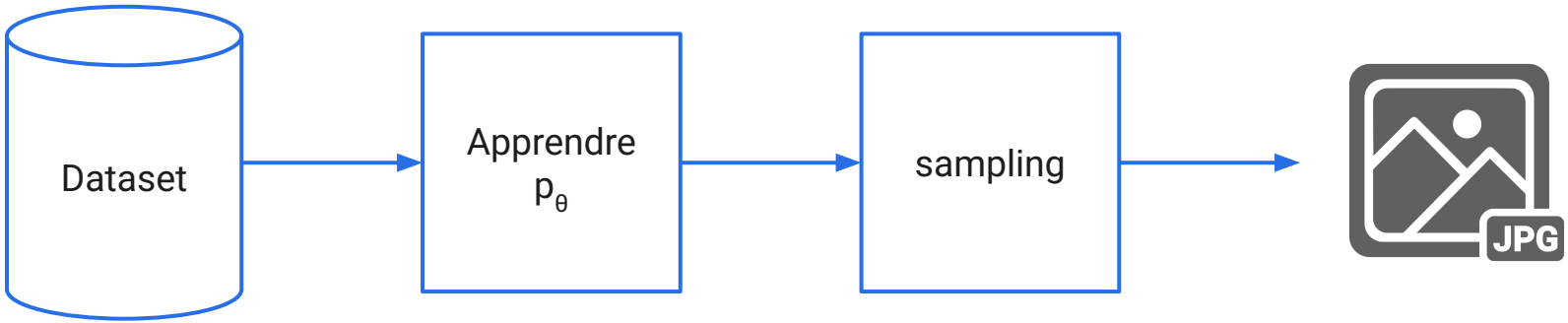
- Données
  - distribution web ≠ distribution métier (branding, packaging, imagerie technique) → risques de dérive
- Propriété intellectuelle / conformité
  - “IP-safe” et traçabilité deviennent des exigences produit (ex. Adobe)
- Sécurité
  - filtres (NSFW), refus, watermark/provenance (selon contexte)
- Fiabilité
  - reproductibilité (seed + versions modèles + scheduler), tests de non-régression visuelle
- Performance
  - VRAM (résolution), latence (steps), coût GPU (batch, precision fp16/bf16)
- Évaluation
  - métriques automatiques + protocole humain (sinon optimisation “à l’aveugle”)
- Résultat attendu
  - un pipeline génératif “pilotable” comme un composant logiciel, pas une démo fragile



Modéliser une distribution d'image

Le problème commun : apprendre une distribution d'images

- On observe des images $x \in \mathbb{R}^{H \times W \times 3}$ (et parfois une condition c)
- Objectif "génératif" : approximer $p_\theta(x)$ (unconditional) ou $p_\theta(x \mid c)$ (conditional)
- Générer = **échantillonner** $x \sim p\theta(\cdot)$ (ou $x \sim p\theta(\cdot \mid c)$)
- c peut être : texte, classe, image source, masque, carte (segmentation / edges / pose), etc.
- Deux notions à ne pas confondre :
  - **training** : ajuster θ pour rapprocher p_θ de p_{data}
  - **sampling** : produire un x (coût/latence ≠ coût d'entraînement)
- Question d'ingénierie : quelle famille donne le meilleur compromis qualité / contrôle / coût pour un usage donné ?



Trois stratégies pour approcher p(x)

- **(A) Densité explicite / latent variable** : on optimise (une borne de) $\log(p_\theta(x))$
  - typique des VAE (log-likelihood *approximée*)
- **(B) Modèle implicite** : on sait sampler $x=G_\theta(z)$, mais pas calculer $p_\theta(x)$
  - typique des GAN (pas de vraisemblance)
- **(C) Processus itératif** : on définit une dynamique qui transforme un bruit en image
  - diffusion / score-based (reverse denoising)
- Coût caché:
  - (A) et (B) peuvent sampler en 1 passe
  - (C) sample en T étapes (scheduler)
- Signal d'apprentissage
  - A) reconstruction + régularisation
  - (B) adversarial
  - (C) prédiction de bruit/score
- Implication pratique
  - les hyperparamètres critiques ne sont pas les mêmes (stabilité vs steps vs guidance)

Conditionnement : respecter des contraintes

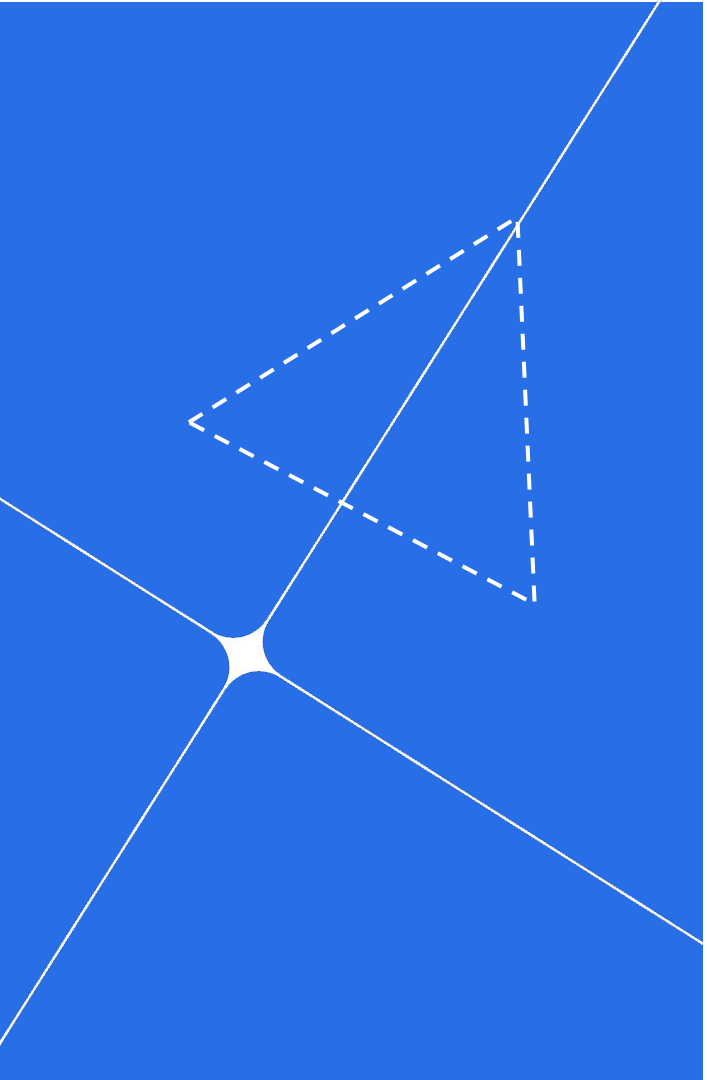
- On passe de $p(x)$ à $p(x \mid c)$: la condition pilote la distribution (pas juste un post-traitement)
- Exemple :
  - c = texte → contraintes sémantiques (“astronaute”, “style photo”, “éclairage”)
  - c = image + paramètre α → interpolation *fidelity vs creativity* (img2img)
  - c = masque → génération **locale** (inpainting) : x_{connu} figé, $x_{manquant}$ généré
- Implémentations typiques du conditionnement :
  - concaténation (channels), FiLM/AdaIN, **cross-attention** (très courant en diffusion texte→image)
- Point clé : “conditionnement” ≠ “contrôle total” → on parle souvent de **soft constraints**

Espace latent : pourquoi on compresse l'image avant de générer

- Générer directement en pixels est coûteux : dimension énorme, textures fines, dépendances longues
- Stratégie moderne : apprendre un encodeur/décodeur (souvent de type VAE) et générer dans z
  - $z=E(x)$, $x \approx D(z)$
- Si diffusion en latent
  - on apprend $p(z \mid c)$ puis on reconstruit $x=D(z)$
- Avantages
  - moins de VRAM, plus rapide, résolutions plus élevées accessibles
- Risque
  - le bottleneck latent peut "jeter" de l'info → artefacts / perte de micro-détails (selon VAE)
- Idée "système" : découpler
  - (1) compression perceptuelle (VAE) et
  - (2) génération stochastique (diffusion)

Du cadre conceptuel au pipeline logiciel

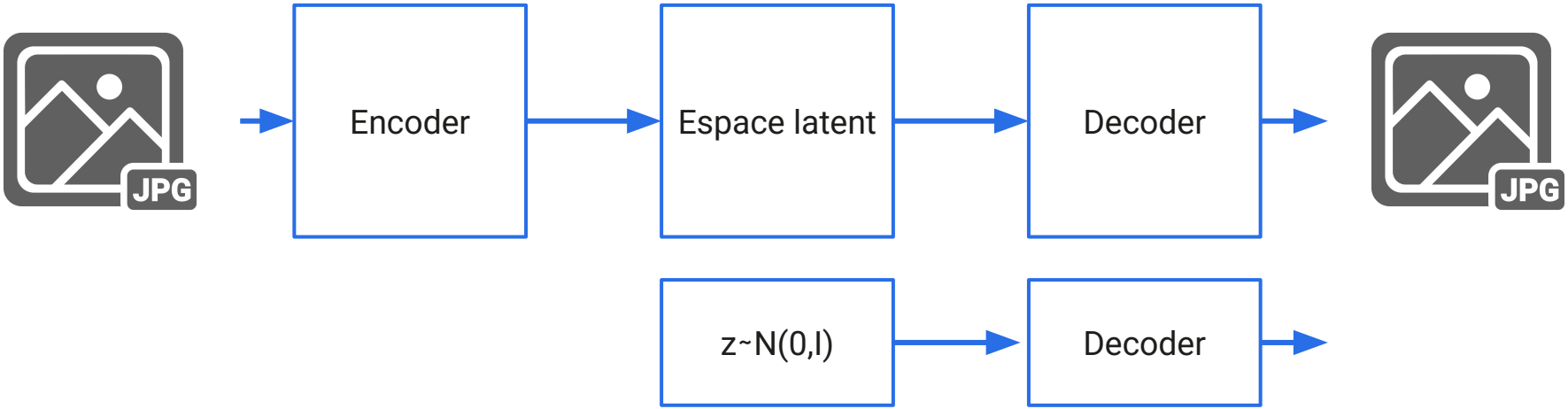
- Un *pipeline* = composants + orchestration du sampling (pas juste “un modèle”)
- Pour diffusion
  - **model** (U-Net), **scheduler** (règles de pas), **conditioning** (encodeur texte), **decoder** (VAE)
- Même modèle, tâches différentes
  - text2img / img2img / inpainting = entrées/contraintes différentes (mêmes briques)
- Paramètres qui changent la distribution échantillonnée
  - seed, steps, guidance, sampler, résolution
- Bon réflexe d’ingénieur
  - logguer tous les paramètres (sinon impossible de reproduire/debugger)
- Transition : on va maintenant détailler ces stratégies via VAE, GAN, puis diffusion



VAE : générer via variables latentes

Pourquoi des VAE (Variational AutoEncoder)?

- Objectif : apprendre un **modèle probabiliste** génératif (≈ maximum likelihood) plutôt qu'un modèle implicite
- Entraînement généralement **stable** (pas de jeu min-max comme les GAN)
- Bonus "ingénieur"
  - on obtient un **espace latent** z exploitable (interpolation, clustering, contrôle simple)
- Génération = "échantillonner un latent" puis décoder (pipeline très clair)
- Les VAE seuls peuvent être "moins nets", mais ils sont une brique clé des pipelines modernes (latent diffusion)
- Question directrice
  - comment concilier **reconstruction** + **régularisation** pour rendre z génératif ?

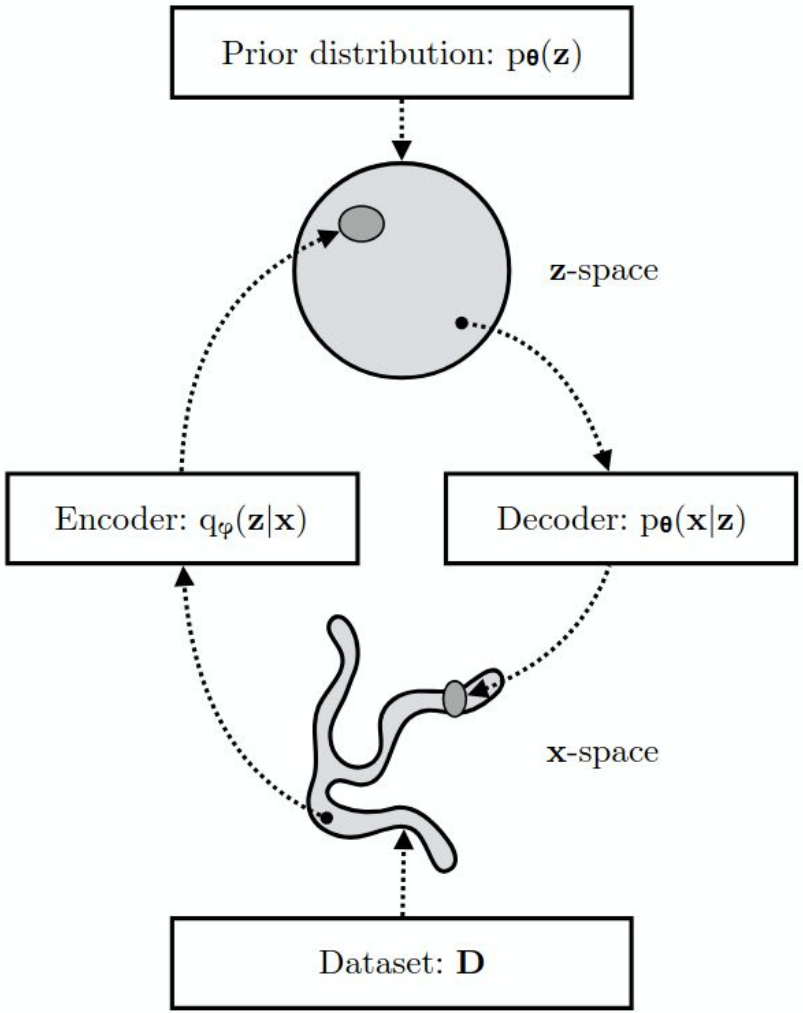


Modèle génératif

- Variables : image x, latent z (dimension $d << H \cdot W \cdot 3$)
- Prior simple :
  - $p(z)=N(0,I)$ (régularise l'espace latent)
- Décodeur (générateur) :
  - $p_\theta(x \mid z)$ (souvent Gaussien ou Bernoulli selon le prétraitement)
  - Gaussien : $p_\theta(x \mid z)=N(\mu_\theta(z), \sigma^2 I)$ où le décodeur (un réseau de neurones) prédit $\boldsymbol{\mu_\theta(z)}$ (une image) et on fixe σ (ou on le paramétrise grossièrement).
  - Bernoulli surtout pour images binaires / jouets (MNIST)
- Problème :
  - $p_\theta(z \mid x)$ (posterior) est **intractable** → on l'approxime
- Approx posterior (encodeur) :
  - $q_\phi(z \mid x)$ (typiquement Gaussien diagonal $q_\phi(z \mid x)=N(\mu_\phi(x), \mathrm{diag}(\sigma^2{}_\phi(x)))$
- En génération : on n'a besoin que de $p(z)$ et $p_\theta(x \mid z)$
- À noter : si on ne génère pas de distributions (Gaussiennes) mais seulement des points, on a un autoencoder classique.

Vue d'ensemble

Kingma, D. P., & Welling, M. (2019). An introduction to variational autoencoders.



Objectif clé : l'ELBO (Evidence Lower Bound)

- On veut maximiser $\log(p_\theta(x))$, mais $\log(p_\theta(x)) = \log(\int_z p_\theta(x|z)p(z)\,dz$ est difficile
- VAE optimise une borne :

$$\log p_\theta(x) \geq \underbrace{\mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x \mid z)]}_{\text{reconstruction}} - \underbrace{\mathrm{KL}(q_\phi(z \mid x)\|p(z))}_{\text{régularisation latent}}$$

- Lecture pratique :
  - reconstruire x depuis z
  - forcer $q(z|x)$ à ressembler à $N(0,I)$
- Si **KL ≈ 0** : *posterior collapse*, z est peu/plus utilisé (le décodeur peut ignorer le latent).
- Si **KL trop grand** : $q(z|x)$ s'éloigne du prior → échantillonner $z \sim p(z)$ donne des générations médiocres (mismatch prior/posterior).
- On souhaite contrôler le compromis reconstruction ↔ régularisation
  - **β-VAE** : multiplier le terme KL par β pour favoriser des latents plus factorisés/interprétables
- Point d'attention : l'ELBO est un compromis, pas une garantie "qualité perceptuelle"

Reparameterization trick : backprop à travers l'échantillonnage

- Encodeur produit $\mu_\phi(x)$ et $\log(\sigma_\phi^2(x))$ (Gaussien diagonal)
- Échantillonner naïvement $z \sim N(\mu, \sigma^2)$ casse la différentiabilité
- Astuce : écrire l'aléa séparément

$$\epsilon \sim N(0,I), \quad z = \mu_\phi(x) + \sigma_\phi(x) \odot \epsilon$$

- On backprop dans μ, σ ; l'aléa est dans ϵ (indépendant des paramètres)
- En pratique : perte = recon_loss+KL ; optimisation SGD/Adam standard
- Résultat : un entraînement "deep learning friendly" malgré les variables latentes

Reparameterization trick : backprop à travers l'échantillonnage

```
# Pseudo-code PyTorch (VAE Gaussien diagonal)
mu, logvar = encoder(x)                    # (B, d), (B, d)
std = (0.5 * logvar).exp()
eps = torch.randn_like(std)
z = mu + std * eps                         # reparameterization
x_hat = decoder(z)

recon = F.mse_loss(x_hat, x, reduction="sum")  # ou BCE selon x
kl = -0.5 * torch.sum(1 + logvar - mu**2 - logvar.exp())
loss = recon + kl
loss.backward()
opt.step()
```

Échantillonnage et conditionnement (CVAE) : générer “sous contrainte”

- Sampling de base :
  - $z \sim N(0,I)$, puis $x \sim p_\theta(x \mid z)$
- Conditionnement c (classe/texte/attributs)
  - viser $p_\theta(x \mid z, c)$ et $q_\phi(z \mid x,c)$
- Intuition : c fixe “quoi générer”, z gère la stochasticité (variantes)
- Pour images
  - CVAE utile pour génération par classes, attributs (ex. “smiling / not smiling”)
- Limite
  - le texte libre est moins naturel à intégrer en VAE “simple” que via cross-attention (diffusion)
- Pont vers la suite
  - en diffusion latente, le VAE sert surtout à **compresser/décompresser**, et le conditionnement texte est géré ailleurs

Limites des VAE et variantes importantes

- “Blurriness” fréquent : vraisemblance pixel simple (Gaussian/Bernoulli) → moyenne de modes visuels
- **Posterior collapse** : un décodeur trop puissant peut ignorer z (KL≈0)
- β-VAE : pousse l’indépendance des facteurs (meilleur “disentanglement”, recon parfois moins bonne)
- **VQ-VAE** : latents **discrets** (vector quantization), réduit certains collapses, ouvre la voie à des priors autoregressifs
- “VAE comme composant système” : dans **Latent Diffusion**, on diffuse dans z (efficacité) puis on décode en pixels
- Message clé :
    - même si la diffusion domine la génération finale, **comprendre le VAE** est indispensable pour comprendre Stable Diffusion / SDXL (qualité, artefacts, VRAM, résolution)

Exemple de sortie d'un VAE (visages)

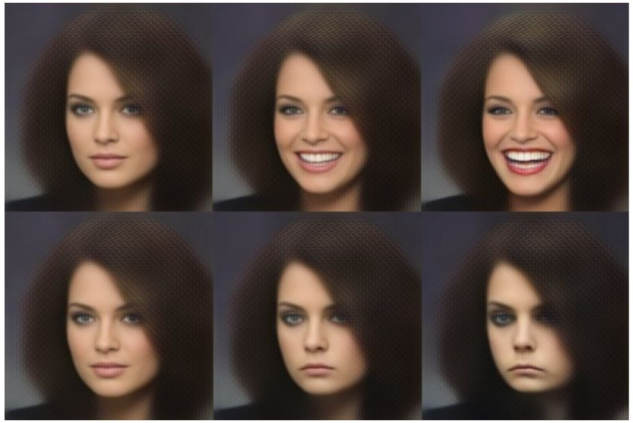
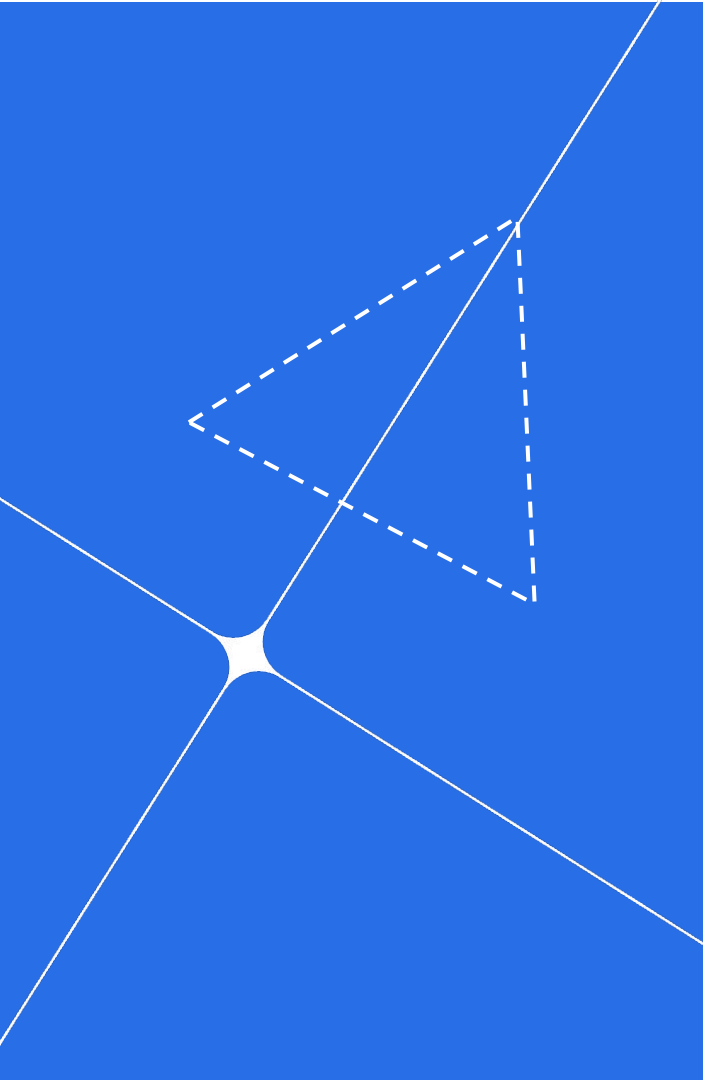


Figure 4.4: VAEs can be used for image resynthesis. In this example by White, 2016, an original image (left) is modified in a latent space in the direction of a *smile vector*, producing a range of versions of the original, from smiling to sadness.

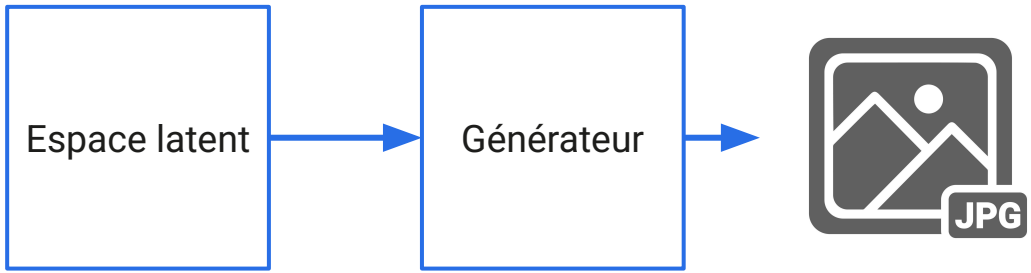
Kingma, D. P., & Welling, M. (2019). An introduction to variational autoencoders.



GAN : génération par jeu adversarial

Pourquoi les GAN (Generative Adversarial Networks) ?

- Motivation computer vision :
  - générer des images **très nettes** (textures, détails) en **une passe** (sampling rapide)
- Modèle **implicite** :
  - on sait générer x=G(z), sans imposer une vraisemblance explicite $p_\theta(x)$
- Très bons résultats sur visages / objets / textures (ex. StyleGAN : contrôle multi-échelle des styles)
- Gros impact produit : génération temps réel / interactive (latence faible vs diffusion itérative)
- Tâches où les GAN brillent :
  - génération d'assets, textures, "style transfer", image-to-image (pix2pix, CycleGAN)
- Mais :
  - entraînement souvent **instable** (min-max), et risque de *mode collapse* (diversité)

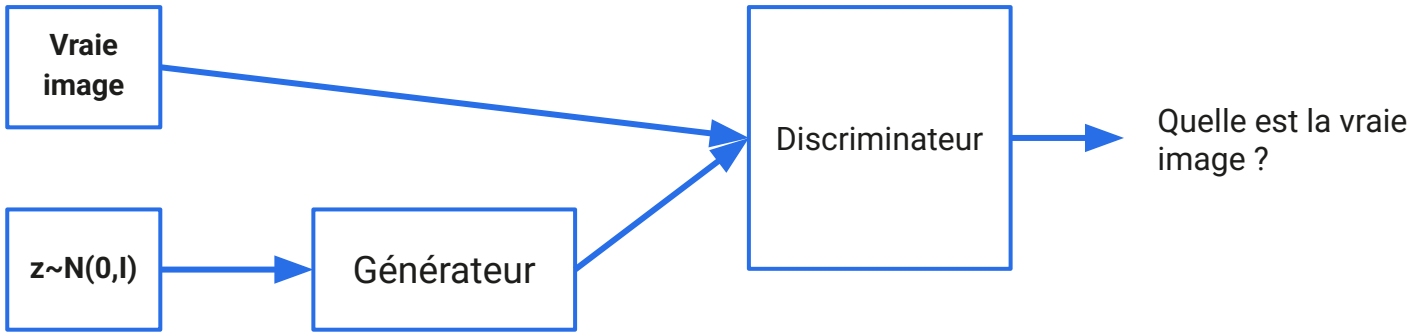


Objectif GAN : min-max entre Générateur et Discriminateur

- Deux réseaux : $G_\theta(z)$ génère, $D_\psi(x)$ discrimine (classification image réelle ou générée)
- Objectif original (Goodfellow 2014) :

$$\min_G \max_D \; \mathbb{E}_{x\sim p_{\text{data}}}[\log D(x)] + \mathbb{E}_{z\sim p(z)}[\log(1 - D(G(z)))]$$

- Intuition : D apprend un **signal d'apprentissage** (gradient) pour pousser G vers la distribution réelle
- On apprend G et D en même temps : le générateur devient meilleur à générer des images, mais le discriminateur s'améliore aussi à détecter la fraude
- En pratique : on utilise souvent la "non-saturating loss" pour G (meilleurs gradients)
- **Échantillonnage** : z~N(0,I) → x=G(z)
- Lecture ingénieur :
  - qualité = architecture + loss + régularisations + dataset (plus que "la formule")



Entraînement : ce qui casse, et les stabilisateurs “classiques”

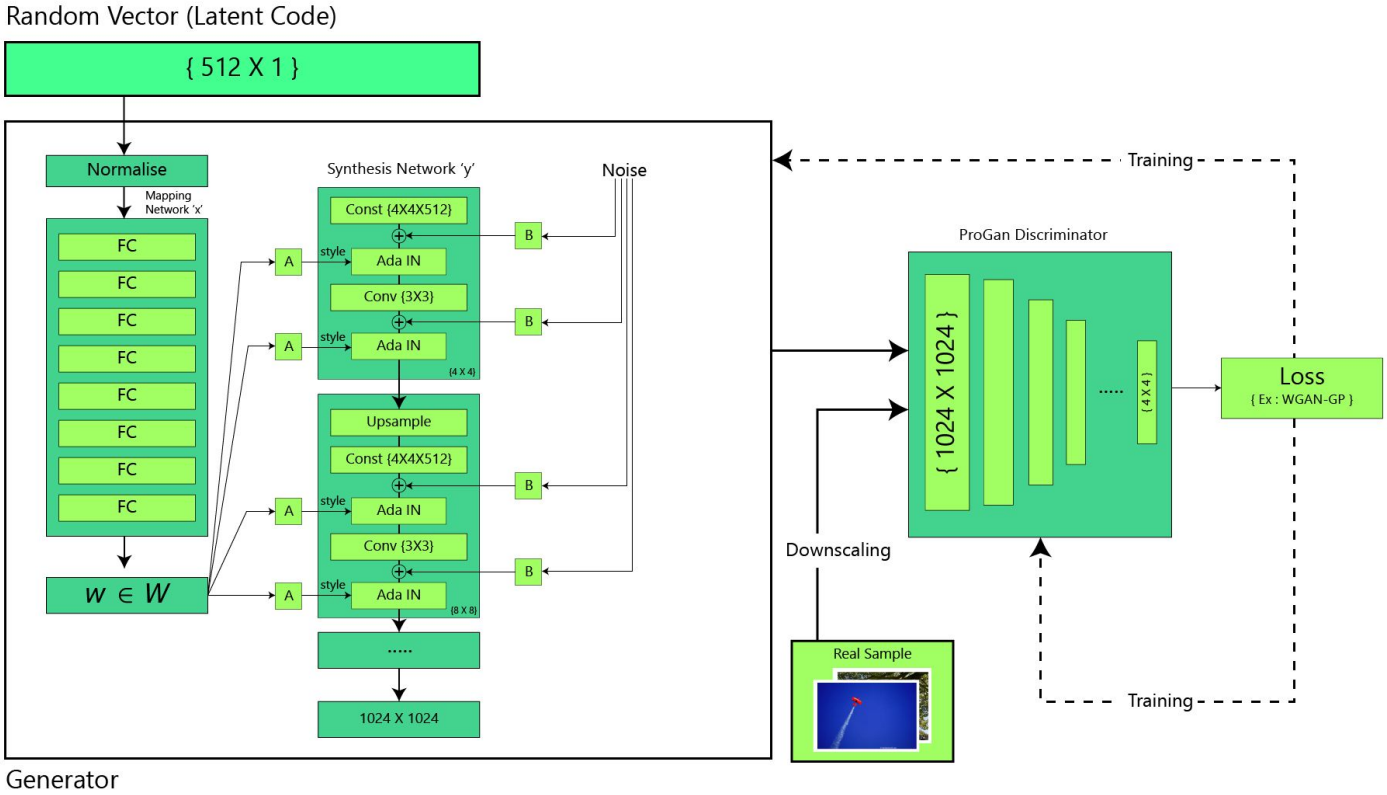
- Pathologies : *mode collapse*, gradients faibles, oscillations (jeu min-max)
- Plusieurs solutions proposées:
  - WGAN : remplacer divergence Jensen-Shannon par distance de Wasserstein + contrainte Lipschitz → gradients plus utiles, apprentissage plus stable
  - WGAN-GP : imposer 1-Lipschitz (|f(x) - f(y)| <= ||x - y||) via gradient penalty (évite le weight clipping)
  - Spectral Normalization : contraindre le Lipschitz du discriminateur simplement, stabilise souvent très bien
  - Pertes modernes : hinge loss (souvent avec SN) ; + “tricks” (TTUR, data aug, etc.) selon contexte
- Message :
  - les GAN sont “sensibles” → il faut traiter l’entraînement comme un problème de **contrôle** (monitoring)

Architectures GAN : de DCGAN à StyleGAN / BigGAN

- DCGAN : règles de design avec des convolutions (strides, BN, ReLU/LeakyReLU) → baseline stable et efficace
- SAGAN : self-attention pour dépendances longues (cohérence globale)
- BigGAN : GAN conditionnel à grande échelle ; **truncation trick** = régler fidélité vs diversité
- StyleGAN / StyleGAN2 : générateur “style-based” (contrôle multi-échelle, mixing) + meilleure qualité perceptuelle
- Industrialisation : modèles + code de référence souvent publiés (ex. dépôts NVLabs)
- À retenir : GAN = très performant, mais tuning/data/compute déterminent énormément le résultat

Architectures GAN : de DCGAN à StyleGAN / BigGAN

- DCG
- SAG
- BigG
- Style
- Indu
- À re



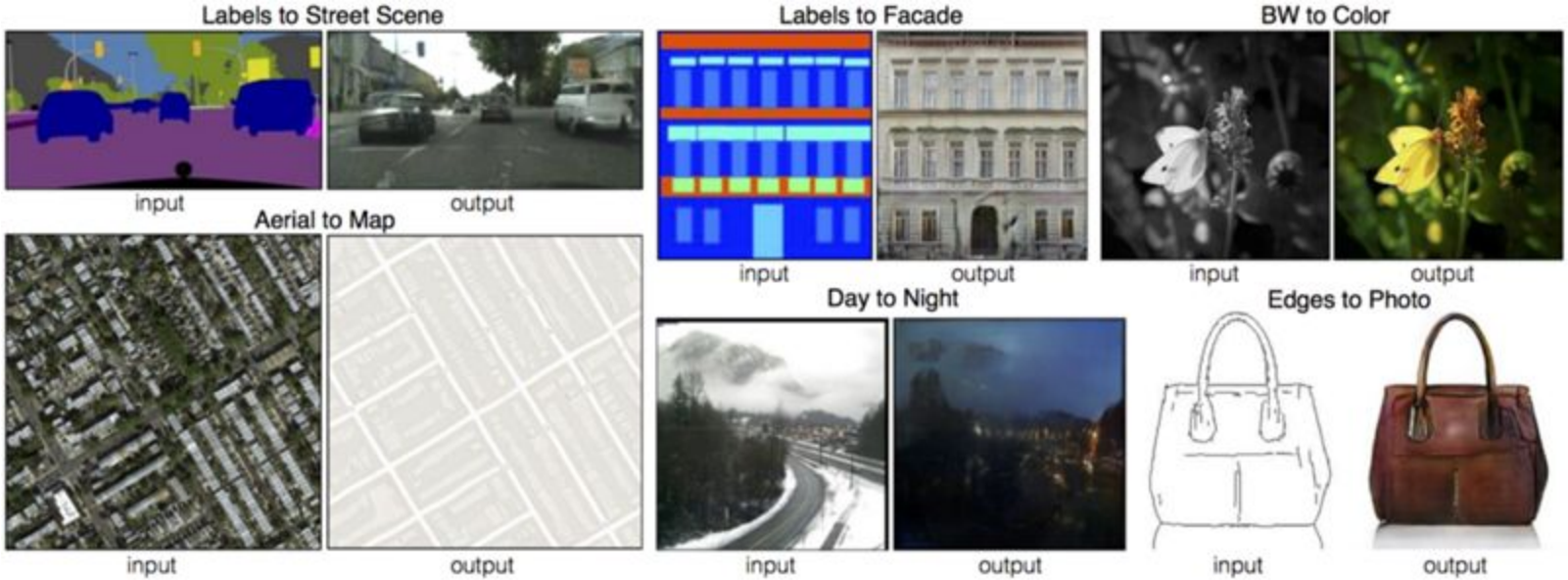
GAN conditionnels pour “image-to-image” : paired vs unpaired

- pix2pix (paired) : apprendre x→y + perte adversariale + (souvent) L1 pour préserver la structure
- Cas d’usage :
  - edges→photo
  - segmentation→photo
  - day→night (quand données alignées disponibles)
- CycleGAN (unpaired) : deux mappings G: X→Y et F:Y→X + **cycle-consistency**

$$\mathcal{L}_{cycle} = \mathbb{E}_x \left\| F(G(x)) - x \right\|_1 + \mathbb{E}_y \left\| G(F(y)) - y \right\|_1$$

- SPADE / GauGAN : génération à partir de layouts sémantiques (contrôle fort via segmentation)
- Lecture produit :
  - ces modèles offrent un contrôle structurel très direct (inputs explicites)
- Limite
  - généralisation hors distribution + artefacts si contraintes trop dures / données trop faibles

GAN conditionnels pour “image-to-image” : paired vs unpaired

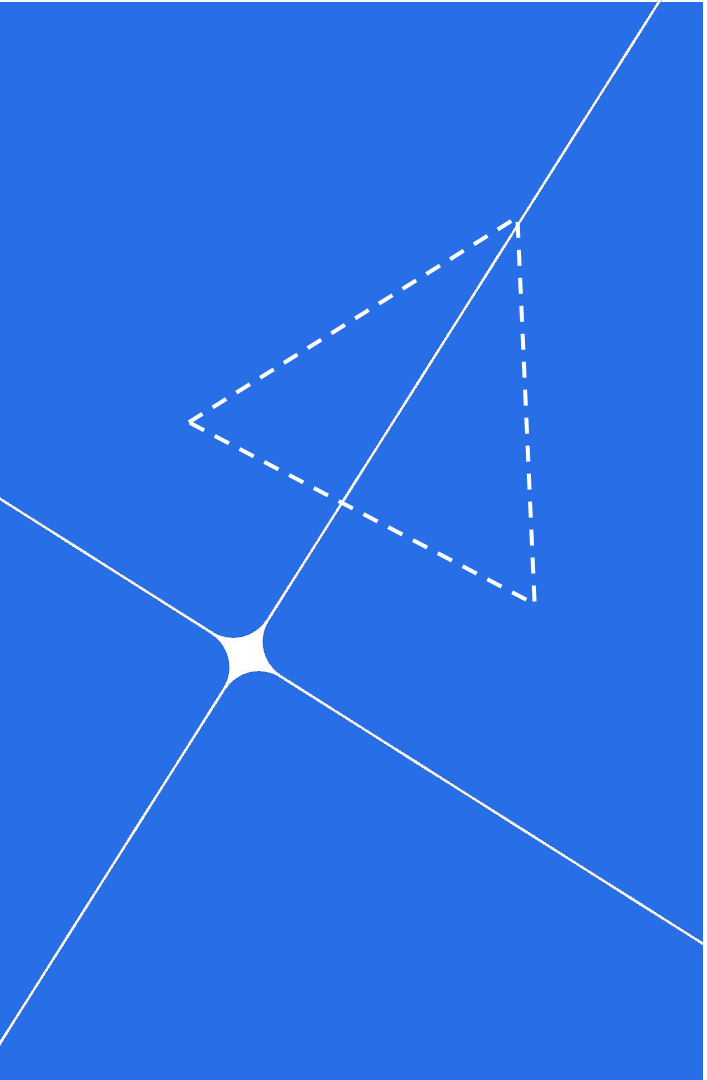


Exemple code : boucle d'entraînement GAN (très générique)

```
# Pseudo-code PyTorch: WGAN-GP (très simplifié)
for x_real in loader:
    for _ in range(n_critic):
        z = torch.randn(B, d, device=x_real.device)
        x_fake = G(z).detach()

        d_real = D(x_real).mean()
        d_fake = D(x_fake).mean()
        loss_D = -(d_real - d_fake)

        # gradient penalty
        eps = torch.rand(B, 1, 1, 1, device=x_real.device)
        x_hat = eps * x_real + (1 - eps) * x_fake
        g = torch.autograd.grad(D(x_hat).sum(), x_hat, create_graph=True)[0]
        gp = ((g.flatten(1).norm(2, dim=1) - 1) ** 2).mean()
        loss_D = loss_D + lam * gp

        opt_D.zero_grad(); loss_D.backward(); opt_D.step()

    z = torch.randn(B, d, device=x_real.device)
    loss_G = -D(G(z)).mean()
    opt_G.zero_grad()
    loss_G.backward()
    opt_G.step()
```

Où placer les GAN en 2026 ?

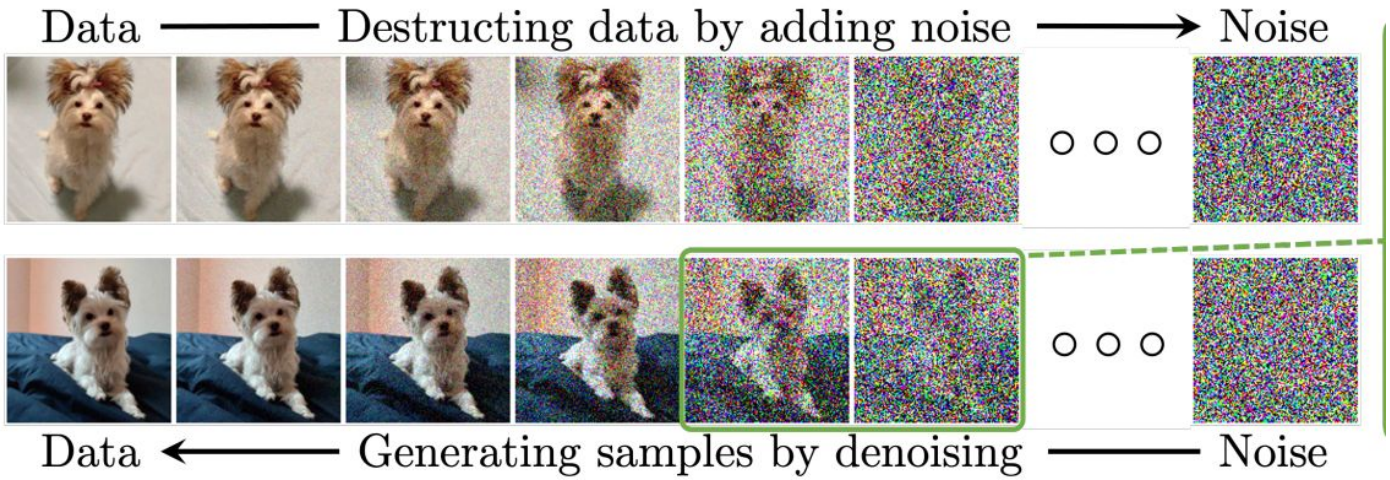
- GAN restent pertinents quand **latence** et **netteté** priment (génération “one-shot”)
- Très bons pour des tâches structurées (segmentation/layout → image) type SPADE/GauGAN
- Très utiles en **augmentation de données** (domaines rares, médical, industriel), mais à valider (biais/artefacts)
- Diffusion domine le text-to-image “open world”, mais les GAN restent une boîte à outils essentielle
- Insight clé :
  - “GAN = sampling rapide, tuning délicat”
  - “diffusion = sampling lent, tuning plus prédictible”
- Transition : on passe aux modèles de diffusion, qui reprennent l’idée de conditionnement mais via débruitage itératif



Diffusion : génération par débruitage

Pourquoi la diffusion a “gagné” pour le text-to-image ?

- Objectif pratique : **qualité + diversité + conditionnement** (texte, masque, image) dans un cadre d’entraînement robuste
- Diffusion = apprendre à **inverser un processus de bruitage** : du bruit vers une image plausible
- Entraînement sans discriminateur : signal de loss simple (souvent MSE) et stable à grande échelle
  - selon les modèles, on prédit ε (bruit), x_0 (image initiale) ou v (velocity), et la loss peut être repondérée
- Contrôle post-training : on peut guider le sampling (CFG, Classifier-Free Guidance, contraintes) sans réentraîner le modèle
- Très compatible avec “génération conditionnée” : texte via cross-attention, contrôles spatiaux, etc.
- Coût principal : sampling **itératif** (T étapes) → latence/compute à gérer (scheduler, steps)

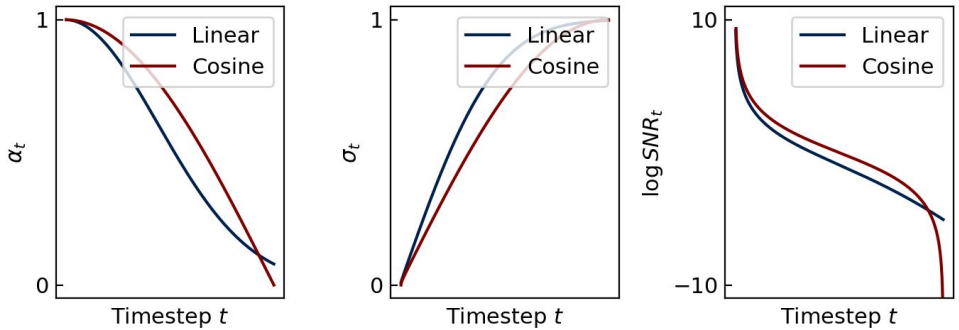


Processus forward : ajouter du bruit de manière contrôlée

- On définit une chaîne markovienne $q(x_t \mid x_{t-1})$ qui ajoute du bruit gaussien
- Paramètres : $\beta_t \in (0,1)$, $\alpha_t = 1 - \beta_t$, $\bar{\alpha}_t = \prod_{s=1\ldots t} \alpha_s$
- Définition typique :

$$q(x_t \mid x_{t-1}) = N(\text{sqrt}(\alpha_t) * x_{t-1}, \beta_t I)$$

- Intuition : β_t = “dose de bruit” au pas t (noise schedule)
- À grand t, x_t perd l’info sémantique et tend vers du bruit
- En diffusion latente, on fait la même chose sur un latent z (pas sur les pixels)



Formule utile : échantillonner x_t directement depuis x_0

- Propriété clé (DDPM, Denoising Diffusion Probabilistic Models) : on peut écrire x_t sans simuler tous les pas
- Formule :

$$x_t = \text{sqrt}(\bar{\alpha}_t)\, x_0 + \text{sqrt}(1-\bar{\alpha}_t)\, \epsilon, \quad \epsilon \sim N(0, I)$$

- Lecture : x_t = mélange “signal original” + “bruit pur” avec un poids dépendant de t
- Conséquence : l’entraînement peut tirer des triplets (x_0, t, x_t) efficacement
- En pratique : on tire t~Uniform{1..T}, puis on construit x_t via la formule
- Cette équation explique pourquoi la diffusion “ressemble” à du denoising multi-niveaux
- Exemple:
  - Si $\bar{\alpha}_t = 0.1$, alors à t, nous avons des coefficients de 0.316 et 0.949

Processus reverse : apprendre à enlever le bruit

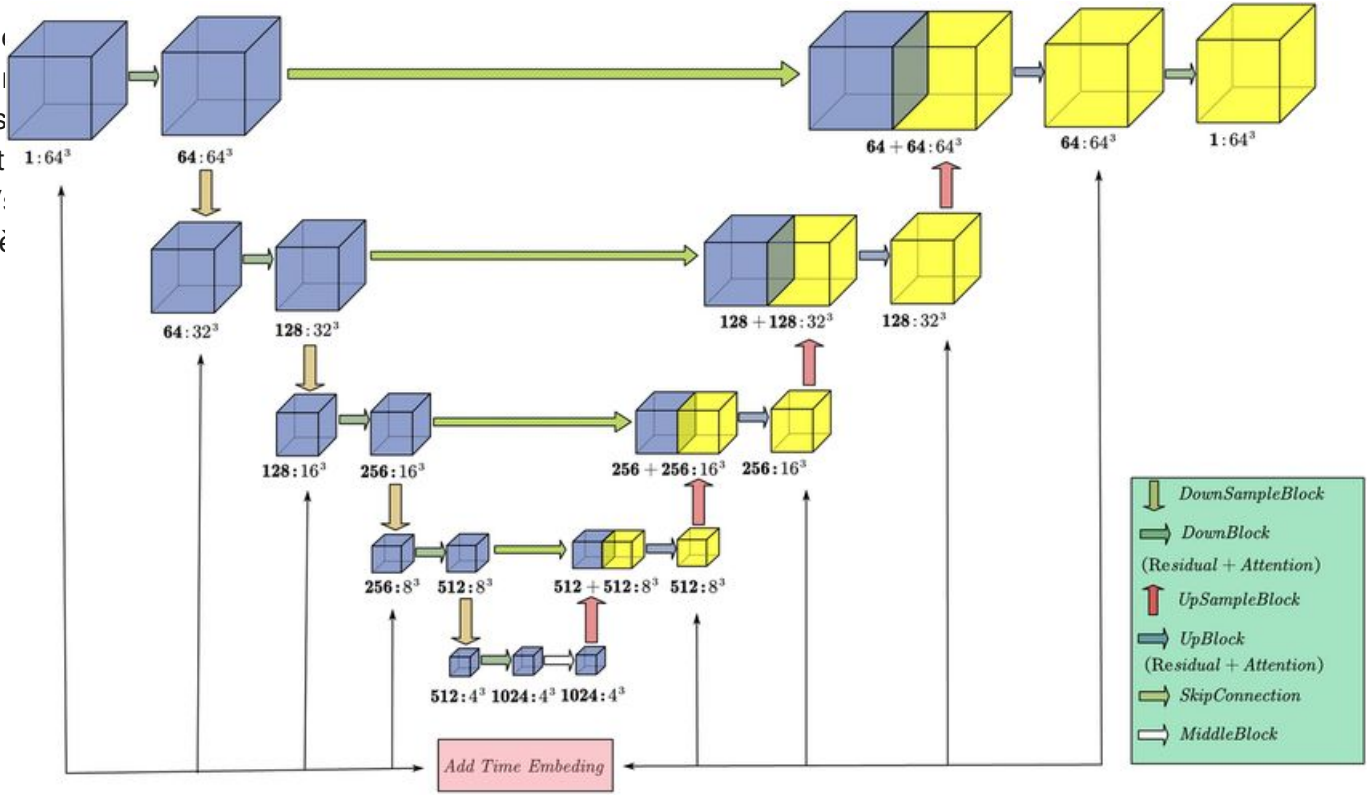
- But : approximer $p_\theta(x_{t-1} \mid x_t)$ pour remonter vers x_0
- On entraîne un réseau $\epsilon_\theta(x_t, t, c)$ à prédire le bruit ϵ injecté (ou x_0)
- Loss standard (version simple) :
  - $L(\theta)=E_{x0,t,\epsilon} \; // \; \epsilon-\epsilon_\theta(xt,t,c) \; //^2_2$
- Pourquoi ça marche
  - si on sait prédire le bruit, on sait reconstruire une direction “vers des images plausibles”
- Sampling : partir de $x_T \sim N(0,I)$ et appliquer un update T fois (scheduler)
- Conditionnement c : texte / image / masque / contrôle spatial (détails plus loin)

Architecture du débruiteur : U-Net + embedding de temps + (souvent) attention

- Backbone courant : U-Net (down / bottleneck / up) + skip connections (détails + structure)
- Embedding du temps t : t→MLP→injection dans les blocks (FiLM-like)
- Blocs résiduels + normalisations : stabiliser l'apprentissage sur des signaux bruités
- Pour text-to-image : insertion de **cross-attention** à plusieurs résolutions
- L'entrée/sortie : tenseur image (pixels) ou latent (typiquement 64×64×4 en LDM, Latent Diffusion Model)
- Vue système : le réseau apprend un "prior" visuel + la capacité à appliquer les contraintes c

Architecture du débruiteur : U-Net + embedding de temps + (souvent) attention

- Backbon
- Embeddi
- Blocs rés
- Pour text
- L'entrée/
- Vue systè

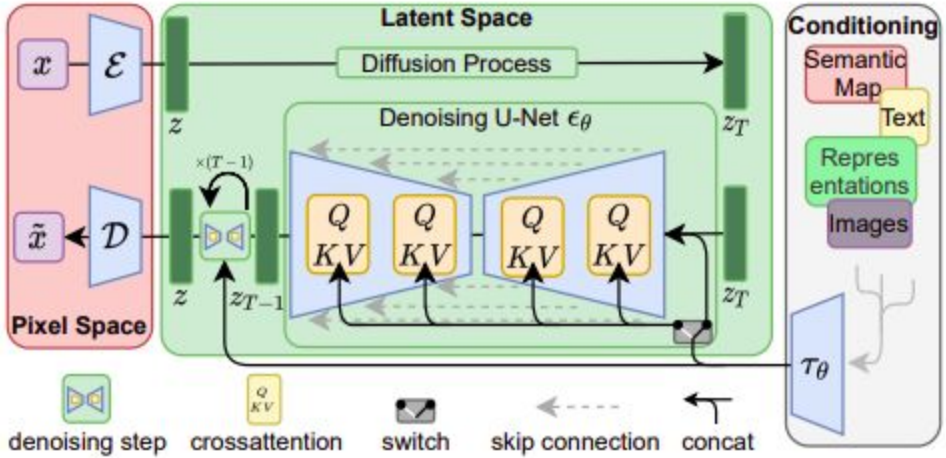


Conditionnement texte : embeddings + cross-attention

- Texte → tokens → embeddings (encodeur texte figé ou semi-figé selon modèles)
- Cross-attention :
  - queries = features U-Net
  - keys/values = embeddings texte => injection sémantique
- Effet : le modèle “décide” où/quoi dessiner en fonction des mots, à plusieurs niveaux (coarse→fine)
- Le conditionnement n’impose pas une contrainte dure : il “biaise” la trajectoire de débruitage
- Extensions :
  - conditionnement spatial (edges, pose, depth) via réseaux additionnels (ControlNet)
  - On réutilise la même mécanique pour img2img / inpainting (avec entrée image/masque)

Conditionnement texte : embeddings + cross-attention

- Texte → tokens → embeddings (encodeur texte figé ou semi-figé selon modèles)
- Cross-attention :
  - queries = featu
  - keys/values = e
- Effet : le modèle “déc →fine)
- Le conditionnement
- Extensions :
  - conditionneme
  - On réutilise la n



Classifier-Free Guidance (CFG) : régler “fidélité au prompt” vs diversité

- Idée : entraîner le modèle à fonctionner **avec** et **sans** condition (dropout de c)
- À l’inférence, on combine deux prédictions :

$$\hat{\epsilon} = \epsilon_\theta(x_t, t, \emptyset) + w \cdot \left(\epsilon_\theta(x_t, t, c) - \epsilon_\theta(x_t, t, \emptyset)\right)$$

- w = *guidance scale*
  - plus haut ⇒ plus “prompt-adherent”, mais risque d’artefacts / moins de variété
- Negative prompt = une façon pratique de définir la branche ‘uncond’ (souvent condition vide, mais on peut la remplacer par une condition ‘à éviter’), ce qui change le biais introduit par CFG.
- Dans Diffusers : guidance active quand guidance_scale > 1
- Message clé :
  - CFG est un **paramètre majeur** de pilotage (qualité perçue ≠ métriques)

Classifier-Free Guidance (CFG) : régler “fidélité au prompt” vs diversité

- Idée : entra
- À l’inférenc

- w = *guidan*
  - plus
- Negative p mais on peut la remplacer
- Dans Diffu
- Message c
  - CFG

Prompt = Tom Cruise with a red suit

Sampling : DDPM vs DDIM et la notion de scheduler

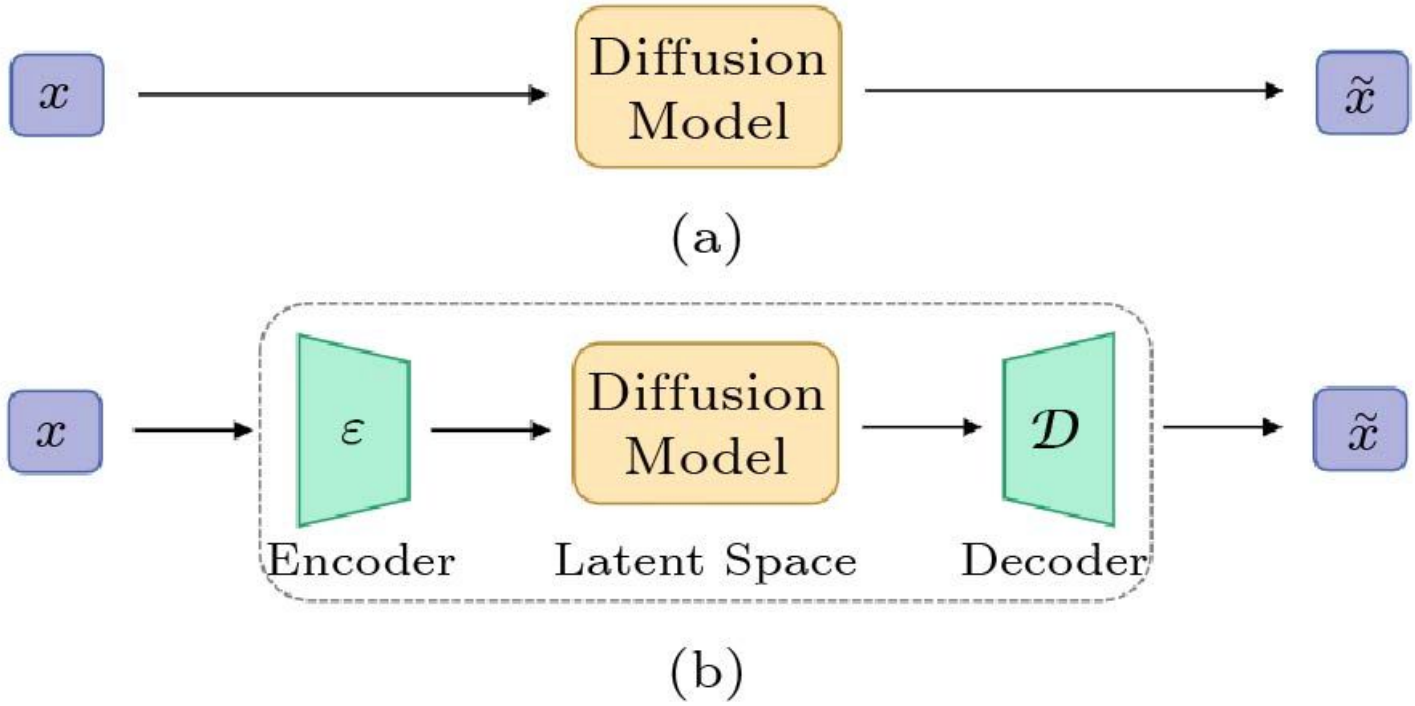
- La génération dépend fortement du **scheduler** (la règle de mise à jour $x_t \rightarrow x_{t-1}$)
- DDPM (Denoising Diffusion Probabilistic Models) : sampling markovien, historiquement plus lent (beaucoup de pas)
- DDIM (Denoising Diffusion Implicit Models) : processus non-markovien (le forward reste un bruitage markovien), sampling accéléré, permet trade-off vitesse/qualité
- Dans la pratique (outils) :
  - Euler/Heun/DPM++ etc. = familles de solveurs numériques (ODE/SDE view)
- Plus de steps (num_inference_steps) ⇒ souvent meilleure qualité, mais plus lent
- Compétence “ingénieur” :
  - traiter scheduler + steps comme des hyperparamètres de production

Diffusion en latent : pourquoi Stable Diffusion est faisable en pratique

- Pixel diffusion (512^2) = coûteux
  - latent diffusion = on débruite z dans un espace compressé
- Pipeline LDM (Latent Diffusion Model) :
  - encoder VAE : $x \rightarrow z$
  - diffusion : bruit $\rightarrow z$
  - decoder VAE : $z \rightarrow x$
- Avantages : VRAM ↓, throughput ↑, résolutions plus hautes accessibles
- Le VAE devient un point critique : s'il perd des détails, la sortie peut avoir un aspect particulier
- LDM ajoute cross-attention pour conditionnements génériques (texte, bbox, etc.)
- Conclusion :
  - Stable Diffusion = diffusion moderne rendue "ingénierable" grâce au latent

Diffusion en latent : pourquoi Stable Diffusion est faisable en pratique

- Pixel diffusion (512^2) = coûteux
  - C
- Pip
  - C
  - C
  - C
- Ava
- Le
- LD
- Co
  - C



Zoom système : Stable Diffusion vs SDXL

- Stable Diffusion = LDM + encodeur texte + U-Net + VAE + scheduler (pipeline modulaire)
- SDXL (Stable Diffusion XL) : U-Net ~3× plus large + plus de blocks d'attention + **2e text encoder**
- SDXL : entraînement multi-aspect ratios (meilleure robustesse de composition)
- SDXL : "refiner" (optionnel) pour débruitage fin en fin de chaîne (qualité perçue augmente)
- Implication pratique
  - meilleurs détails à 1024², mais coût VRAM/latence plus élevé (pipeline plus lourd)

Évolution de Stable diffusion

Exemple minimal (Diffusers) : text-to-image “reproductible”

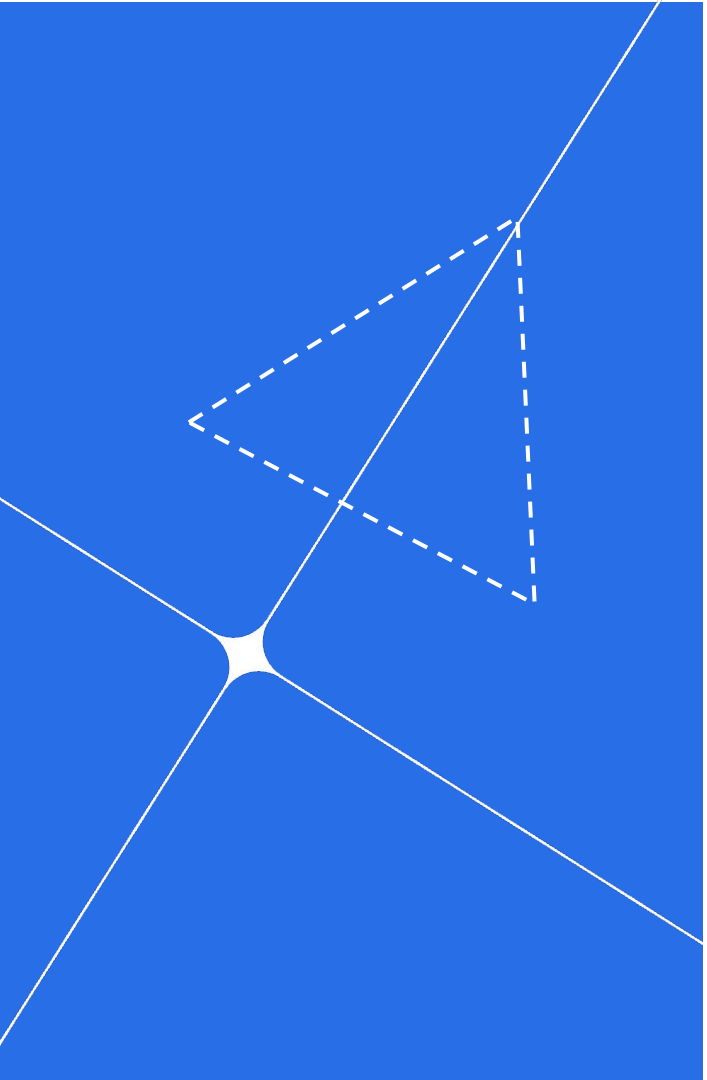
- Pipeline standard : from_pretrained + to("cuda") + génération avec seed + steps + guidance
- Paramètres à toujours tracer
  - modèle, scheduler, seed, num_inference_steps, guidance_scale, résolution
- negative_prompt : spécifier ce qu’on veut éviter (artefacts, styles indésirés)
- Attention : mêmes prompts ≠ mêmes images si versions/schedulers/precisions diffèrent (non-régression)
- Objectif : transformer une démo en composant “testable” (logs + seeds + configs)

Exemple minimal (Diffusers) : text-to-image “reproductible”

```python
import torch
from diffusers import StableDiffusionPipeline

model_id = "runwayml/stable-diffusion-v1-5"  # exemple
pipe = StableDiffusionPipeline.from_pretrained(model_id, torch_dtype=torch.float16)
pipe = pipe.to("cuda")

g = torch.Generator("cuda").manual_seed(42)

img = pipe(
    prompt="photo réaliste d'un renard vert dans une forêt brumeuse, lumière douce",
    negative_prompt="mauvaise anatomie, texte, watermark, flou",
    num_inference_steps=40,
    guidance_scale=7.5,
    generator=g,
    height=512, width=512,
).images[0]
```

Exemple mini ductible”

```
import torch
from diffusers import

model_id = "runwayml/s
pipe = StableDiffusion                                    t16)
pipe = pipe.to("cuda")

g = torch.Generator("c

img = pipe(
    prompt="photo réal                                    uce",
    negative_prompt="m
    num_inference_step
    guidance_scale=7.5
    generator=g,
    height=512, width=
).images[0]
```



“Piloter” un pipeline : text2img, img2img, inpainting, super-res, editing

Le pipeline comme API mentale

- Un pipeline diffusion = **modèle** (U-Net), **scheduler**, **conditionneur** (texte/image), **VAE decode/encode**
- Text2Img : on initialise un latent bruité z_T, on débruite → z_0, puis decode → image
- Img2Img : on encode une image $x \rightarrow z_0$, on ajoute du bruit (niveau contrôlé), puis débruitage conditionné
- Inpainting : on conserve la partie non masquée, on ne génère que la zone manquante (mais avec cohérence globale)
- Super-res / Upscale : soit modèle dédié SR, soit stratégie “tile + diffusion” pour préserver détails
- Editing : pilotage fin = combiner prompt + image + masque + paramètres (pas “un seul bouton”)
- Règle d’or : pour debugger, on change **un paramètre à la fois** (sinon impossible d’attribuer la cause)

Les paramètres principaux

- **Seed** : reproductibilité ; même seed ≠ même image si modèle/scheduler/precision changent
- **num_inference_steps** : +steps ⇒ souvent +qualité, -steps ⇒ +vitesse (mais attention aux artefacts)
- **guidance_scale (CFG)** : +guidance ⇒ +adherence prompt, mais risque d'artefacts / saturation
- **scheduler/sampler** : apparence et stabilité différentes (Euler vs DDIM vs DPM++), à choisir selon objectif
- **résolution** : coût ~H×W (en latent aussi) , avec surcoûts liés aux blocs d'attention ; attention aux ratios non vus à l'entraînement
- **negative_prompt** : utile pour éliminer défauts récurrents (texte, watermark, anatomie)
- **batch / num_images_per_prompt** : exploration vs coût ; utile pour sélectionner (human-in-the-loop)

Text-to-image : bonnes pratiques “production-like”

- Fixer une **template** de prompt (contenu, style, contraintes) + champs variables (produit, ambiance)
- Définir une plage raisonnable : steps (20–50), guidance (5–9) pour SD1.5/SDXL (selon modèle)
- Toujours logguer : modèle exact + revision/sha, scheduler, steps, guidance, seed, résolution, prompt complet
- Pour la diversité : générer k variantes (seeds différentes) puis sélectionner via heuristique ou humain
- Si le style doit être constant : préférer LoRA/adapter plutôt que “prompt-only” (sinon instable d’un seed à l’autre)
- Anti-pattern : guidance très élevée + peu de steps → images “cramées” / artefacts
- Vérifier systématiquement : mains, textes, logos, géométrie (zones de faiblesse connues)

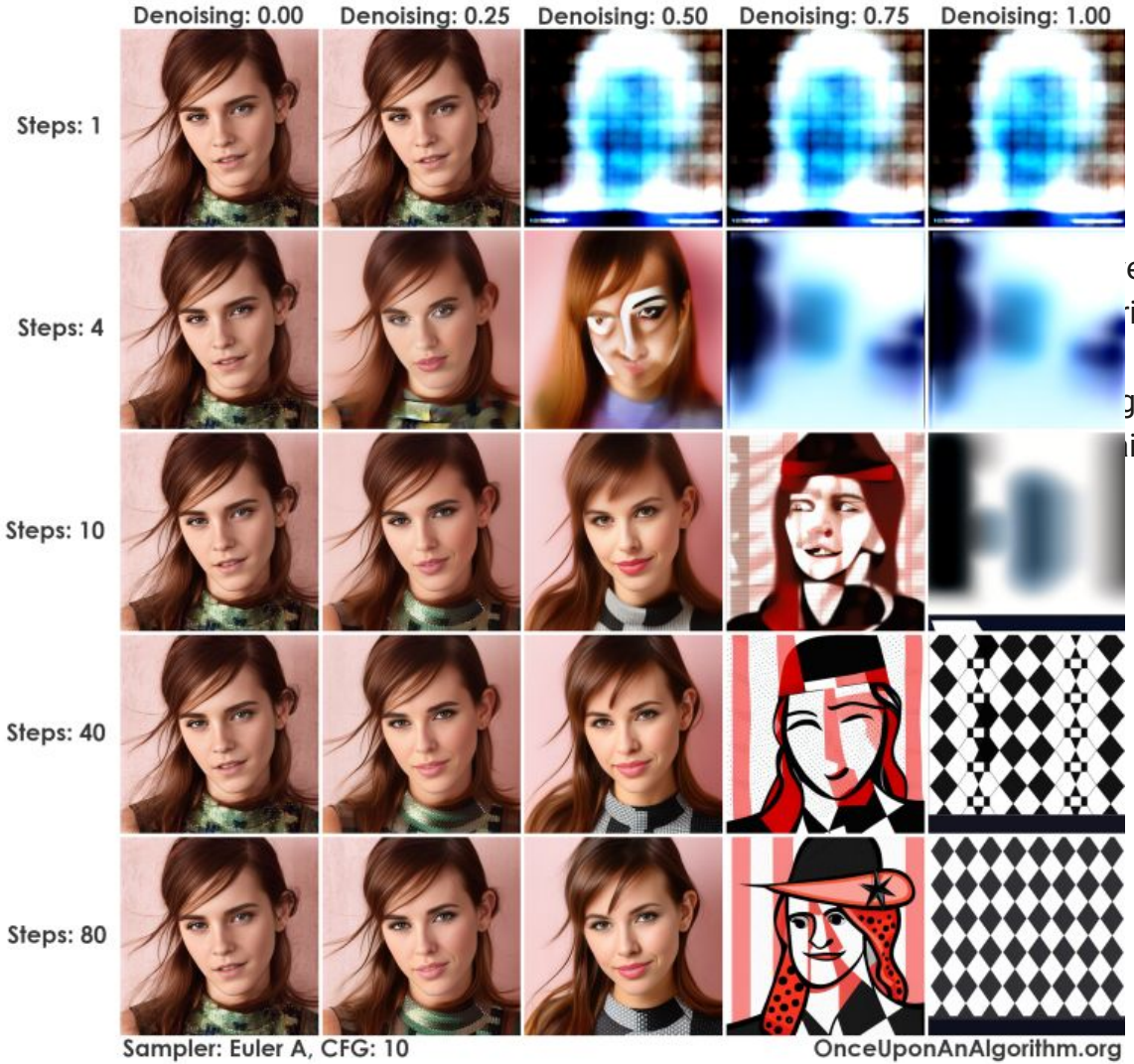
Img2Img : le paramètre “strength” (ou noise level)

- Pipeline : encode image → latent z, ajouter bruit jusqu’à un niveau t, débruiter conditionné → nouveau z
- Paramètre clé : **strength** ∈[0,1] (ou “denoising_start/end”) : contrôle combien on “s’éloigne” de l’original
- Strength bas (0.2–0.4) : conserve composition/identité, changements subtils (couleurs, style)
- Strength haut (0.6–0.9) : plus créatif, mais structure peut dériver (nouveaux objets, pose qui change)
- Bon usage : *style transfer moderne* (rendu artistique), variations de packshot, harmonisation d’éclairage
- Debug typique :
  - “ça ne change pas assez” ⇒ augmenter strength ou modifier prompt
  - “ça change trop” ⇒ baisser strength
- Attention : selon scheduler, strength ↔ nombre effectif de steps (interactions non triviales)

Img2Img : I

- Pipeline : encoc
- Paramètre clé :
- Strength bas (0
- Strength haut ((
- Bon usage : *sty*
- Debug typique :
  - “ça ne ch
  - “ça chang
- Attention : selo

eau z
iginal
ge)
irage



Inpainting / Outpainting : cohérence locale vs globale

- Entrées : image + **masque** (1 = zone à régénérer, ou l'inverse) + prompt (souvent)
- Inpainting : le modèle doit respecter
  - (1) contexte autour du masque
  - (2) contraintes du prompt
  - (3) style global
- Bonnes pratiques masque :
  - bords légèrement **feather** (adoucis) pour éviter une couture visible
  - inclure un peu de contexte dans la zone régénérée (masque légèrement plus large)
- Prompting : décrire ce qui doit apparaître **dans la zone**, pas tout le reste déjà présent
- Outpainting : étendre l'image (ajout de bordures vides + masque)
- Artefacts fréquents : répétition de motifs, incohérences de lumière, "patch" de texture non aligné
- Pour contrôle fort :
  - combiner avec contrôle structurel (edges/depth/pose) via ControlNet (si dispo)

Inpainting / Outpainting : cohérence locale vs globale

Inpainting / Outpainting : cohérence locale vs globale

Super-resolution / Upscaling : gagner en taille sans halluciner

- Objectif : augmenter résolution tout en conservant contenu (et éviter “d’inventer” des détails faux)
- Options pratiques :
  - modèles SR dédiés (diffusion SR, ESRGAN-like) pour fidélité contrôlée
  - pipeline diffusion avec “tile” (découper en patches) pour très haute résolution
- Risque : les modèles génératifs peuvent **halluciner** des détails (médical/technique = dangereux)
- Bon protocole :
  - (1) upscale modéré (×2)
  - (2) vérifier zones critiques (texte, logos, pièces mécaniques)
- En e-commerce/marketing : hallucinations parfois acceptables (tant que visuellement plausibles)
- En imagerie technique : privilégier SR déterministe/physique ou valider avec un expert
- Toujours documenter : méthode SR, facteur d’upscale, seed/paramètres (reproductibilité)

Super-resolut s halluciner

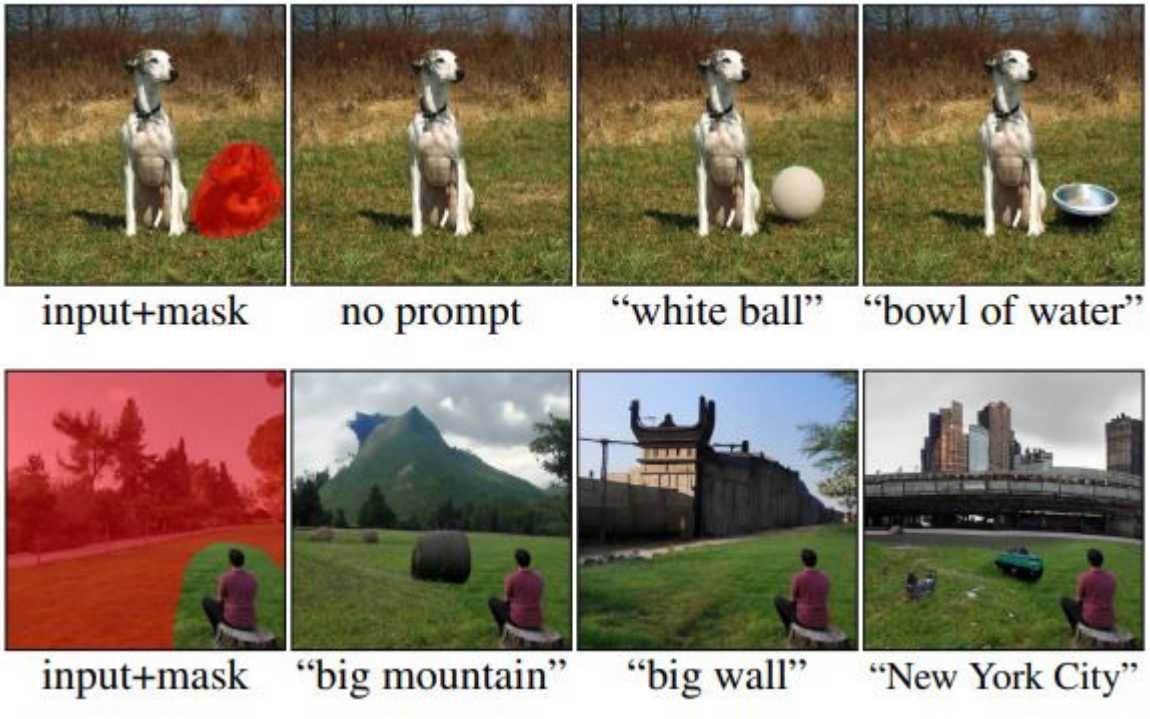
- Objectif : augmente aux)
- Options pratiques :
  - modèles SR d
  - pipeline diffus
- Risque : les modèle ux)
- Bon protocole :
  - (1) upscale m
  - (2) vérifier zor
- En e-commerce/ma les)
- En imagerie techniq
- Toujours document

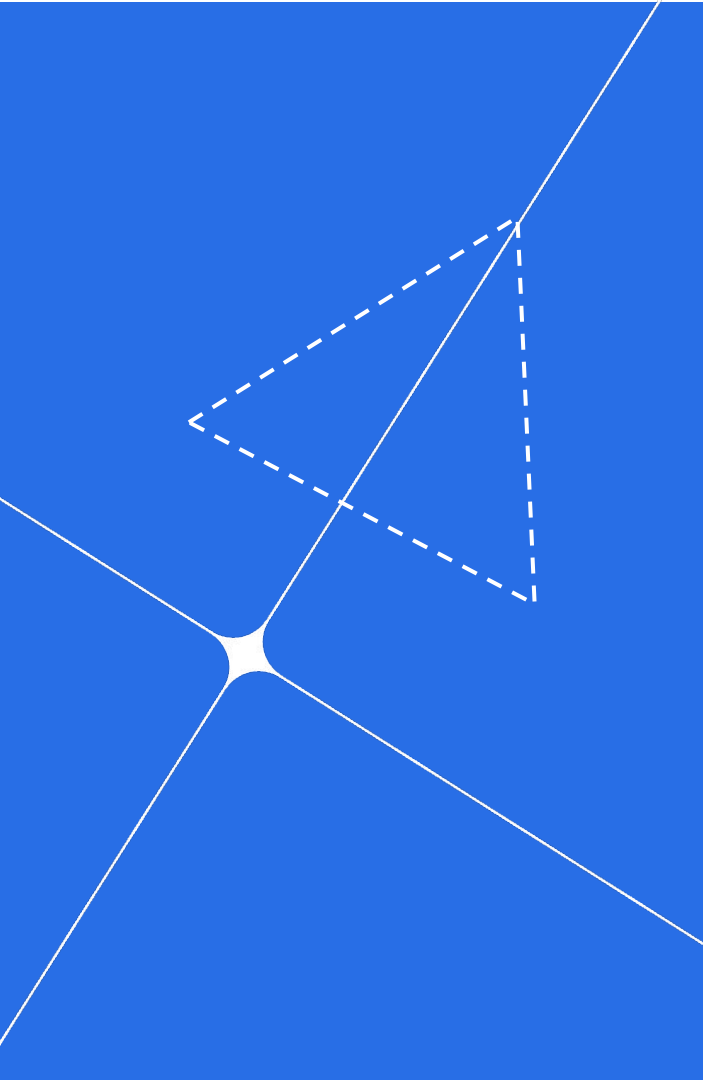
Editing guidé : “modifier X sans casser le reste”

- Cas typiques : changer arrière-plan, couleur d’objet, retirer un élément, ajouter un accessoire
- Approche simple : inpainting local + prompt précis + seed fixe pour itérer
- Approche plus contrôlée : img2img faible strength + masque pour préserver structure
- Garder l’identité/style :
  - limiter la génération à la zone utile
  - éviter prompts globaux contradictoires
- Méthode de debug : isoler d’abord la zone, réussir le local, puis élargir le masque si nécessaire
- Mesurer :
  - comparer avant/après sur des critères explicites (couleur, forme, absence d’artefact)
- Transition : une fois qu’on sait piloter, il faut **évaluer** correctement (métriques + protocole humain)

Editing guidé : “modifier X sans casser le reste”

- Cas typiques : (
- Approche simp
- Approche plus
- Garder l’identité
  - limiter la
  - éviter pro
- Méthode de de
- Mesurer :
  - compare
- Transition : une





Évaluation, reproductibilité et debug

Pourquoi évaluer est difficile (et indispensable)

- Sortie générative = variable
  - même prompt peut donner plusieurs images plausibles → pas de “vérité terrain” unique
- Objectif réel = compromis entre **fidélité visuelle**, **diversité**, **alignement au prompt** (prompt adherence)
- Une seule métrique ne suffit pas
  - certaines corrèlent mal avec le jugement humain sur le text-to-image.
- En production, l’évaluation sert à
  - (1) comparer versions de modèles
  - (2) prévenir régressions
  - (3) détecter dérives
- Deux niveaux : **distribution-level** (qualité globale d’un lot) vs **sample-level** (qualité d’une image)
- Trois familles d’évaluations : métriques automatiques, protocoles humains, tests “contraintes” (mains, texte, comptage)
- Bon réflexe : définir un “benchmark interne” de prompts typiques + edge-cases, stable dans le temps (versionné)

Distribution-level : FID / KID / Precision-Recall (qualité vs couverture)

- **FID** (Fréchet Inception Distance) : compare stats (moyenne/covariance) de features Inception (embedding obtenu en prenant un vecteur intermédiaire dans le modèle Inception) entre réel et généré.

$$\mathrm{FID} = \|\mu_r - \mu_g\|_2^2 + \mathrm{Tr}\left(\Sigma_r + \Sigma_g - 2(\Sigma_r\Sigma_g)^{1/2}\right)$$

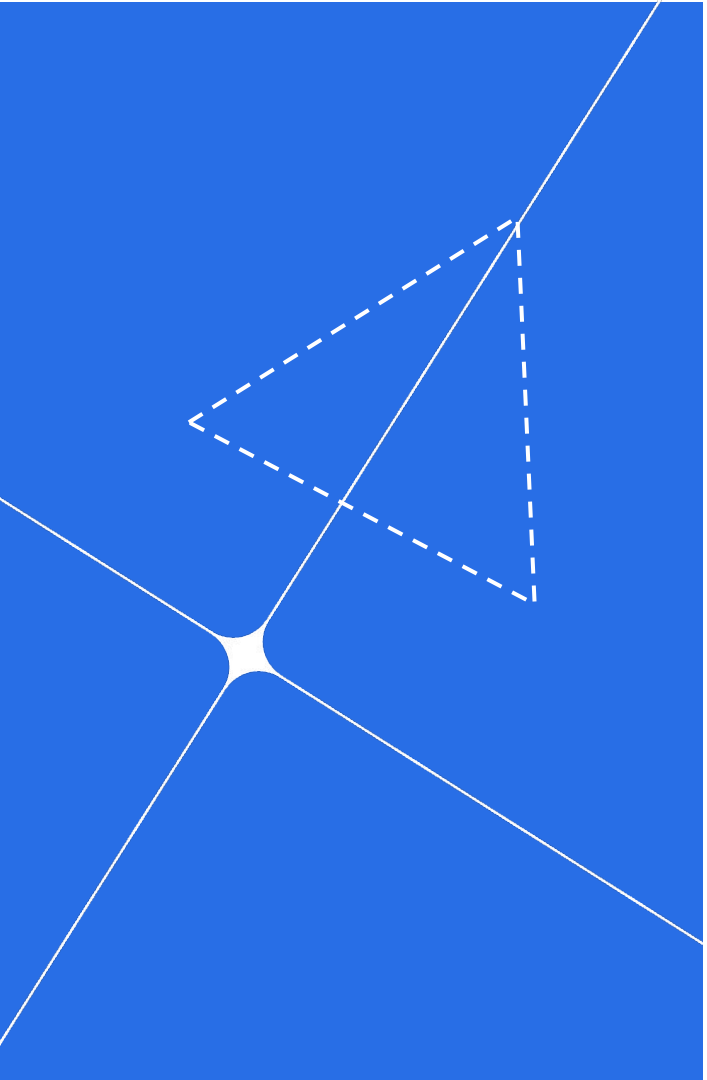
- Lecture : FID décroit ⇒ échantillons générés “plus proches” de la distribution réelle (réalisme + diversité)
  - FID corrèle souvent avec qualité/couverture **sur un protocole fixe**, mais n’isole pas la diversité et dépend fortement du domaine et du nombre d’échantillons.
- **KID** : MMD (Maximum Mean Discrepancy = mesure différence distribution à partir d’échantillons) dans l’espace Inception, estimateur (souvent) moins biaisé sur petits échantillons.
- **Precision/Recall pour générateurs** :
  - sépare “qualité” (precision) et “couverture” (recall)
  - utile pour diagnostiquer *mode collapse*.
  - différent des métriques de classification : calculées à partir d’échantillons (images réelles vs images générées) + comparaison dans un espace de features
- Pièges : dépendance au dataset de référence, au nombre d’échantillons, et au backbone de features (Inception ≠ domaine métier)
- Conseil pratique : rapporter *au minimum* (FID ou KID) + (precision/recall) sur un même protocole fixe

Alignement au prompt + similarité perceptuelle (sample-level)

- **CLIPScore** : similarité image-texte via embeddings CLIP (modèle texte + image), pour estimer l'alignement sémantique (référence-free).
  - Limite : CLIPScore peut être haut avec des images esthétiquement mauvaises (artefacts), donc à combiner avec une métrique "visuelle"
- **Human Preference / HPS** : score appris à partir de choix humains (pairwise preferences) pour mieux refléter ce que les humains préfèrent.
- Pour éditions / restorations : **LPIPS** (distance perceptuelle apprise) utile quand on a une référence "avant/après" ou des variantes à comparer.
- Pour qualité de reconstruction (plus classique) : **MS-SSIM** (structure multi-échelle), surtout utile en compression/restauration.
- En text-to-image "open world", la mesure la plus fiable reste souvent une **évaluation humaine** structurée (rubric + pairwise).
- Message clé :
  - "bon score" ≠ "bonne image" → on garde un œil sur des défauts connus (anatomie, texte, géométrie)

Reproductibilité & debug : rendre la génération testable

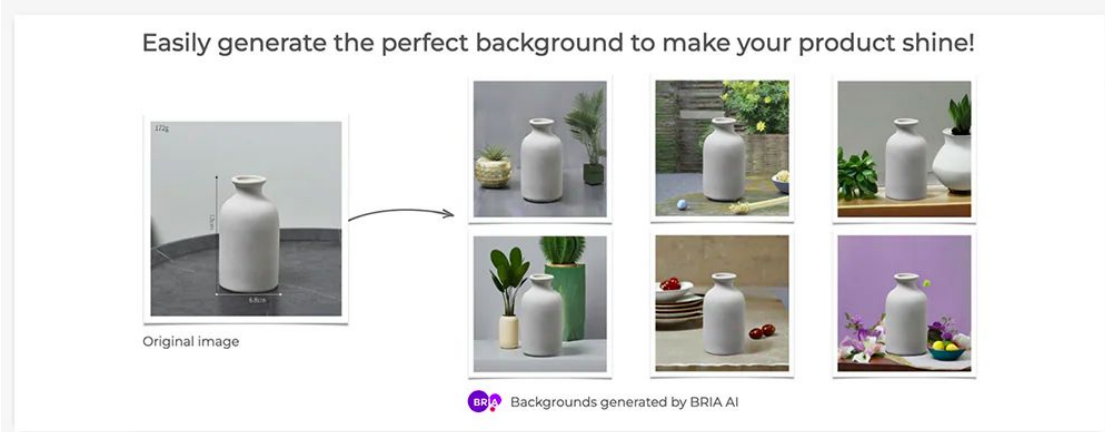
- En diffusion, la reproductibilité dépend de : **seed**, modèle exact, scheduler, steps, guidance, résolution, précision (fp16/bf16)
- Avec Diffusers : passer un **torch.Generator** au pipeline ; attention, le Generator est “consommé” (stateful).
- Protocole simple : sérialiser une config (YAML/JSON) + enregistrer les prompts + seeds + hash/revision du modèle
- Debug efficace : ablations contrôlées (changer 1 paramètre) + sauvegarder les latents intermédiaires (si nécessaire)
- Tests “non-régression” : même set de prompts/seeds, comparer
    - CLIPScore moyen
    - FID/KID sur lot
    - audit visuel
- Évaluation humaine reproductible : privilégier pairwise, consignes claires, randomisation, et traçabilité des prompts/images.
- Anti-pattern : “ça marche sur ma machine” => sans versioning des poids/scheduler, vous ne pouvez pas expliquer une régression



Cas d’usage concrets et patterns produit

E-commerce & marketing : “content ops” à grande échelle

- Problème business : produire **beaucoup** de variantes visuelles (A/B tests, saisons, pays) sans multiplier les shootings
- Pattern “packshot → scènes” : détourage + remplacement de fond (studio, lifestyle, saisonnier)
- Exemple Shopify : génération/remplacement de backgrounds directement dans l’éditeur média (scene presets + prompt)
- Pattern produit : **générer N variantes**, puis sélectionner (humain ou heuristique : netteté, texte parasite, cohérence)
- Extension “site complet” : génération guidée de pages / visuels à partir de keywords (accélère la mise en ligne)
- Évaluation en contexte : KPI online (CTR, conversion), + garde-fous (logos, texte, claims) avant publication
- Risques : incohérences produit (couleur), artefacts “trompeurs” → nécessité d’un check qualité + traçabilité

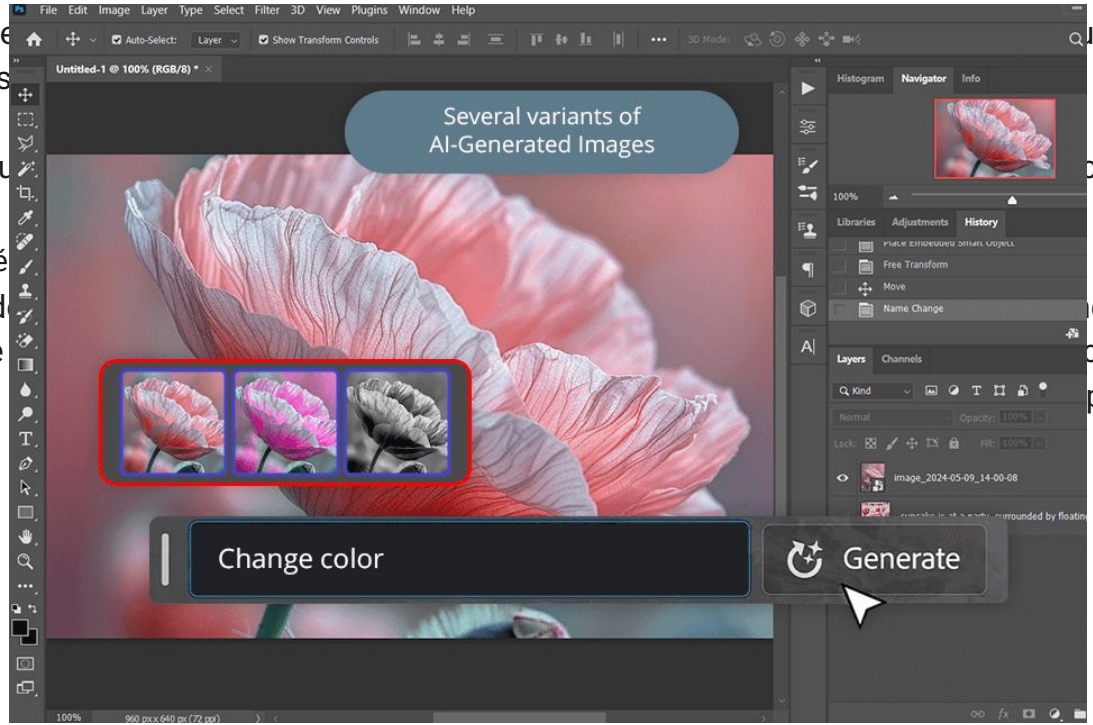


Suites créatives : intégration au workflow

- Pattern clé : la valeur vient de l'**intégration** (Photoshop/éditeur Canva/outil temps réel), pas seulement du modèle
- Adobe Firefly : positionnement "commercially safe" (données sous licence + domaine public) + pas d'entraînement sur contenu client
- Canva : "Magic Studio / Magic Media" = génération d'images directement dans l'éditeur (itérations rapides + mise en page)
    - Signal stratégie : acquisition de Leonardo.Ai par Canva (accélérer l'innovation en "visual AI")
- NVIDIA Canvas : doodles/segmentation en paysages réalistes (concept exploration, backgrounds)
- Pattern entreprise : **modèles custom / brand style** (entraîner sur assets internes) pour cohérence visuelle
- Pattern confiance : provenance/étiquetage via Content Credentials / C2PA (métadonnées "tamper-evident")

Suites créatives : intégration au workflow

- Pattern clé : la vale…lement du modèle
- Adobe Firefly : pos… pas d'entraînement sur contenu client
- Canva : "Magic Stu…ons rapides + mise en page)
  - Signal straté…
- NVIDIA Canvas : d…ds)
- Pattern entreprise …ce visuelle
- Pattern confiance …per-evident")

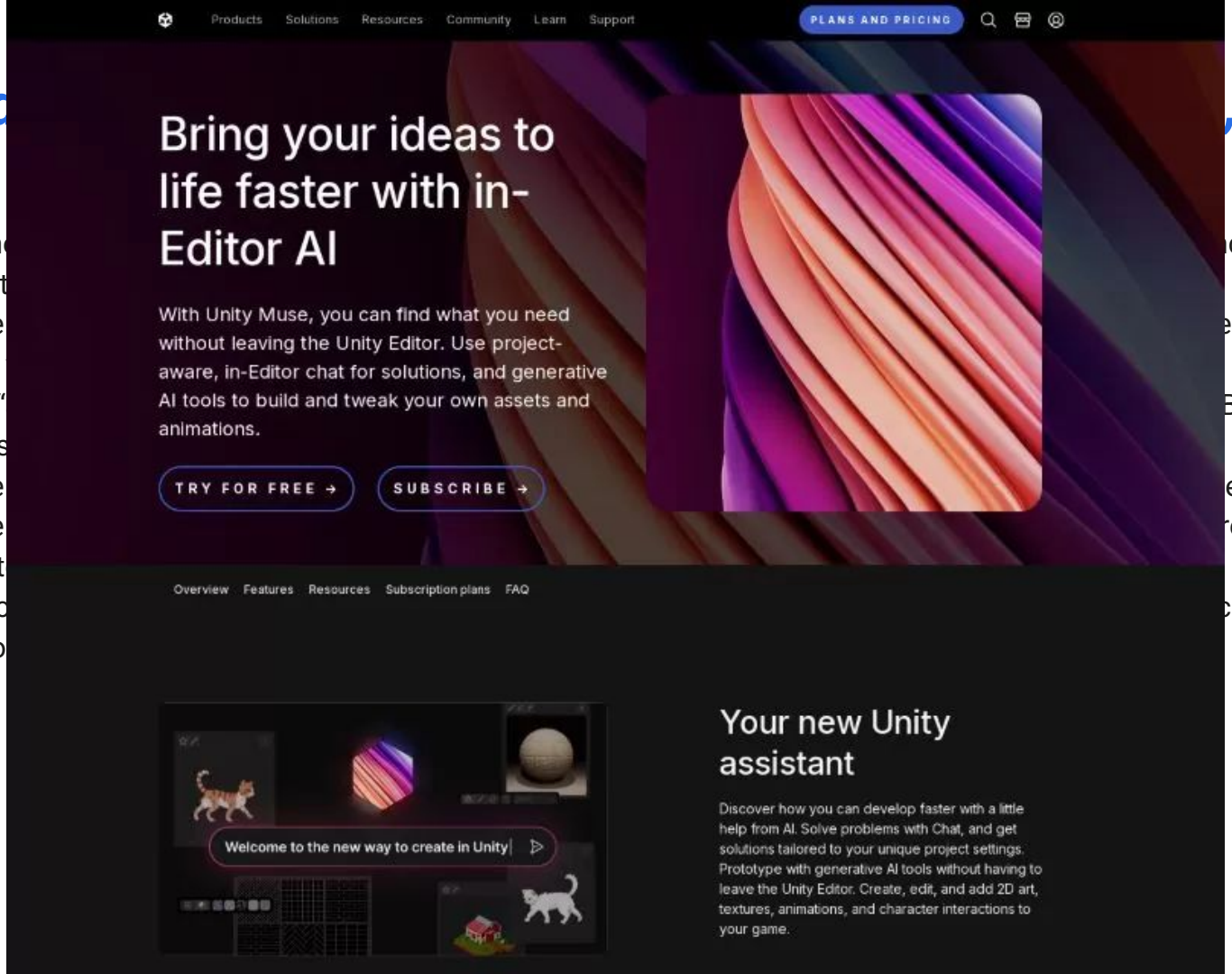


Jeux vidéo / 3D : génération d'assets contrôlés (textures, sprites, concept)

- Problème prod : itérer sur **énormément** d'assets (textures PBR (Physically-Based Rendering), sprites, concepts) avec contraintes d'IP + cohérence
- Exemple Ubisoft La Forge : prototype open-source de **diffusion** entraînée sur assets internes pour générer des textures tileables + conditioning (sketch/height maps)
- Pattern "artist-in-the-loop" : IA pour exploration/variantes, puis retouche humaine + validation pipeline (PBR, tiling, channels)
- Exemple Unity Muse : génération de textures/sprites dans l'écosystème Unity, avec discours "useful and ethical"
- Exemple Blizzard (reporté) : outil interne "Blizzard Diffusion" pour concept art (assisté, pas forcément livré tel quel)
- Pattern technique : contrôler par **inputs structurés** (layouts, masques, maps) plutôt que "prompt-only"
- Point produit : droits & acceptabilité (internes/communautaires) → définir clairement ce qui est "référence", "placeholder", "livrable"

Jeux vid… sprites,

- Problème… …cepts) avec contraint…
- Exemple… …er des textures …
- Pattern “… …BR, tiling, channels…
- Exemple… …ethical”
- Exemple… …é tel quel)
- Pattern t…
- Point pro… …ce”, “placeho…

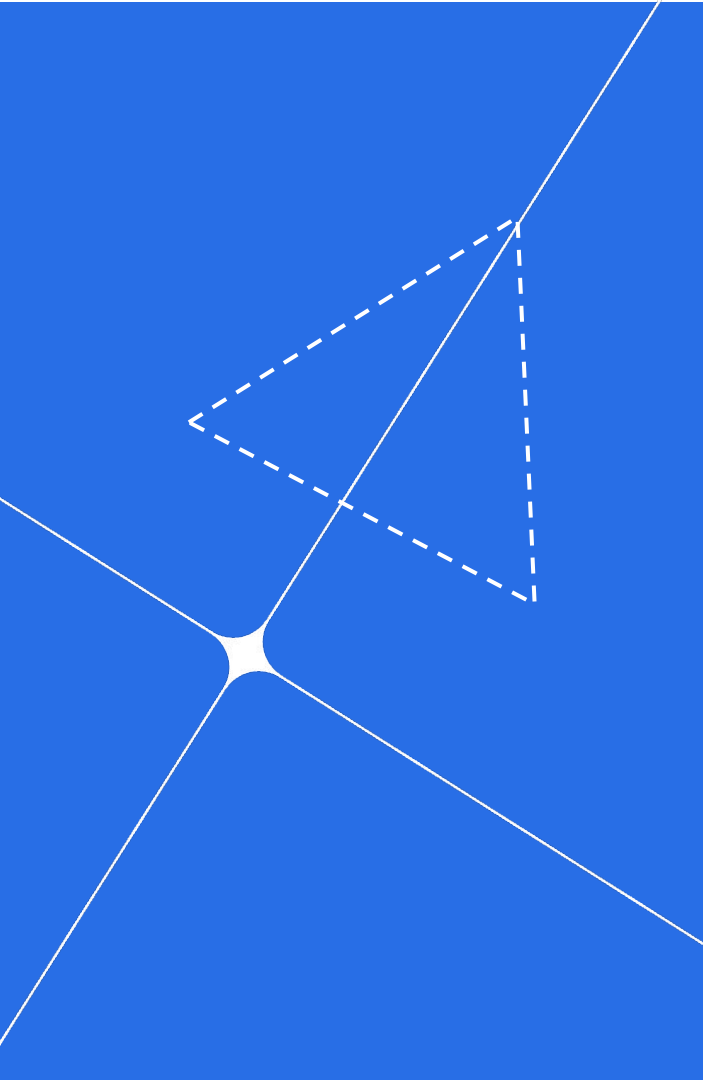


Industrie / QA / médical : données synthétiques et rareté des cas

- Problème : défauts rares, peu d'images annotées, contraintes de confidentialité => modèle de vision sous-entraîné
- Pattern "synthetic defects" : générer des défauts variés (rayures, fissures, manques) + labels parfaits (segmentation/boxes)
- Exemple Edge Impulse : génération d'images synthétiques **labellisées** pour manufacturing (défauts simulés)
- Exemple NVIDIA (médical) : génération de données synthétiques pour augmenter diversité / scénarios difficiles
- Pattern d'évaluation : le bon critère = **gain downstream** (mAP, recall défauts, taux de faux négatifs), pas "images jolies"
- Risque : "synthetic gap" (données trop propres) → domain randomization, calibration, validation sur réel
- Usage diffusion/GAN : génération ciblée (rare cases) + anonymisation partielle, mais audit biais indispensable

Patterns produit transverses : du modèle au service “pilotable”

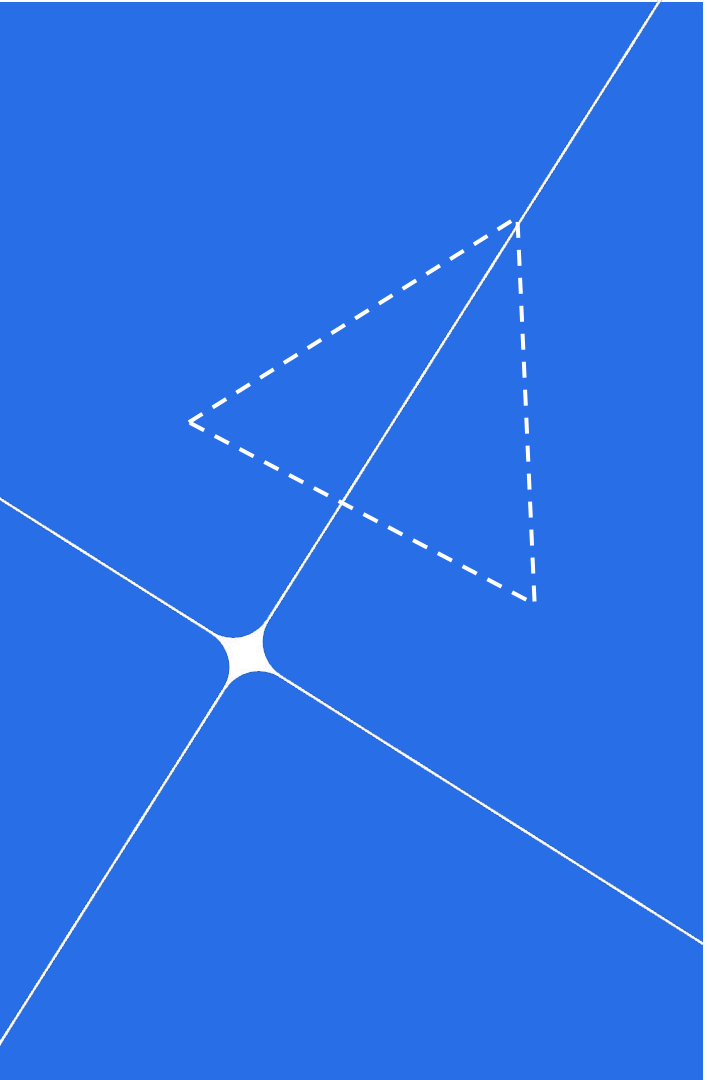
- Pattern UX : **templates de prompts** + champs structurés (produit, style, fond, lumière) → réduit variance utilisateur
- Pattern contrôle : boutons standards (seed, steps, guidance, strength, masque) + presets par tâche (text2img/img2img/inpaint)
- Pattern sélection : “generate → rank → approve” (grille variantes, favoris, history) + logs complets (repro/debug)
- Pattern coût : budgets GPU (steps max, résolution max), batching, files d’attente, priorités (latence vs throughput)
- Pattern conformité : provenance (Content Credentials/C2PA), règles de publication, audit des assets sensibles
- Pattern qualité : tests de non-régression (benchmark prompts internes) + évaluation humaine pairwise sur cas clés
- Pattern sécurité : filtres contenus, “negative prompts” standards, politiques d’usage (ce qui est autorisé/risqué)



Conclusion

À retenir

- **VAE** : cadre probabiliste + latent exploitable ; stable ; qualité parfois “lissée” ; brique clé (compression) en latent diffusion
- **GAN** : sampling **one-shot** très net ; mais entraînement min-max instable + risque *mode collapse* ; très utile pour pipelines contrôlés (image-to-image, textures)
- **Diffusion** : débruitage itératif ; qualité + diversité + conditionnement texte efficace ; knobs majeurs = steps/scheduler/CFG/seed
- Checklist “pilotage” : fixer seed → choisir scheduler → régler steps → ajuster guidance/strength → vérifier artefacts → logguer config
- Checklist “évaluation” : (FID/KID + precision/recall) sur lot + (CLIPScore) + audit humain pairwise sur prompts clés
- Checklist “prod” : versioning modèles + config dump + non-régression sur benchmark interne + provenance/credentiels si diffusion externe



En route vers le TP