## **1. Transformer (the foundational architecture)**

### **What is a Transformer?**

* The Transformer is a deep learning model architecture introduced in the paper *“Attention Is All You Need”*
* It's designed for processing sequences (e.g. text) and models relations between elements (tokens) using attention mechanisms, rather than recurrence (RNNs) or convolutions.
* It consists of two main parts in the original version: an **encoder** stack and a **decoder** stack.
* Over time, variants emerged: encoder‑only (e.g. BERT), decoder‑only (e.g. GPT), and encoder-decoder (e.g. original Transformer, T5).

#### **Key components / mechanisms**

1. **Self-Attention / Scaled Dot-Product Attention**   - For each token, the model computes queries (Q), keys (K), and values (V) via learned linear transforms.  
      - Attention scores = softmax(QKᵀ / √d\_k) gives weights for how much each token attends to each other token.  
      - The output is the weighted sum of the V vectors.  
      - This lets every position see (attend to) all other positions in the sequence.
2. **Multi-head Attention**   - Instead of doing one attention, you use multiple “heads” in parallel (each with its own Q, K, V projections) so you can capture different types of relationships in different subspaces.
3. **Positional Encoding / Positional Information**   - Because attention is permutation-invariant (it treats tokens symmetrically), you need to encode token order.  
      - Positional encodings (e.g. sinusoidal, learned) are added to input embeddings so the model “knows” token positions.
4. **Feed-Forward Networks**   - After attention, each token’s vector is passed through a small feed-forward network (applied identically across positions) to do non-linear transformation.
5. **Residual Connections + Layer Normalization**   - Each sub-layer (attention, feed-forward) has a residual (skip) connection, then normalization, making training more stable.
6. **Encoder / Decoder stacks**   - In the original Transformer:  
       • Encoder: multiple identical layers of (self-attention + feed-forward)  
       • Decoder: similar, but its attention is masked (so you don’t attend to future tokens) plus cross-attention to encoder outputs.
7. **Masking (in decoder side)**   - In autoregressive generation, when predicting token *t*, you can’t peek at future tokens. So you mask out later positions in attention, ensuring only previous tokens are attended to.

Because of all this, Transformers can process sequences in parallel (unlike RNNs) and scale well with large data and compute.

**Use cases of Transformer architecture broadly**

* Machine translation
* Text summarization
* Question answering
* Language modeling / generation
* Protein sequence modeling, DNA, etc.
* Also being used in vision models (Vision Transformers), multimodal models, etc.

## **2. GPT (Generative Pre-trained Transformer)**

Now, with the transformer architecture in place, let’s see how GPT is built and works.

### **What is GPT?**

* GPT stands for **Generative Pre-trained Transformer**.
* It is a family of large language models (LLMs) by OpenAI (GPT‑1, GPT‑2, GPT‑3, GPT‑4, etc.).
* It’s **decoder-only**: uses (a stack of) Transformer decoder layers (with causal/masked self-attention) to generate text in an autoregressive fashion.
* Its training is in two general phases:  
     • **Pre-training**: train on large unlabeled text corpora, with an objective such as predicting the next token.  
     • **Fine-tuning / alignment**: further refinement (often via supervised learning, reinforcement learning from human feedback) on more task-specific or aligned data.

### **How GPT works (internals / algorithm)**

* **Tokenization**: input text is broken into tokens (words, subwords) via a tokenizer (e.g. BPE).
* **Embedding + positional encoding**: tokens get embedded into vectors plus positional info.
* The input is fed into multiple decoder layers, each with masked self-attention (you cannot attend to future tokens) and feed-forward modules.
* The output at each position is projected to vocabulary size, softmax gives probability distribution over next token.
* During generation (inference), GPT picks a token (e.g. via sampling, beam search), appends it to the input, and repeats to generate longer text.

A subtle optimization: when generating one token at a time, you don’t need to recompute everything from scratch. GPT can cache key/value projections from previous steps and only compute for new tokens.

Because GPT is “pre-trained,” it already learns a lot of grammar, facts, world knowledge, and some reasoning patterns. Then, in fine-tuning / alignment, it’s steered to produce more acceptable/safe/useful outputs.

### **Architecture & size**

* GPT‑1 used ~117M parameters.
* GPT‑2 scaled much bigger (hundreds of millions to billions) using more layers, heads.
* GPT‑3 has ~175B parameters (publicly known version).
* GPT‑4 and further versions are multimodal (text + image) and architecture details are less publicly disclosed.

### **Building / implementation details**

* Usually implemented in frameworks like PyTorch or TensorFlow.
* Use attention optimization, parallelism, mixed-precision (FP16), pipeline/model parallelism, etc.
* The training uses massive compute clusters (GPUs, TPUs).
* Techniques such as LoRA (low-rank adaptation) can help in fine-tuning large models by freezing large parts of the model and only training small matrices.

### **Use cases of GPT**

* Chatbots (e.g. ChatGPT)
* Text generation (stories, articles)
* Summarization
* Question Answering / retrieval augmented generation
* Translation, paraphrasing
* Code generation (via Codex, described below)
* Many other NLP tasks (classification, completion)

### **Strengths and limitations**

**Strengths**

* Very flexible and general-purpose
* Learns rich patterns from large-scale data
* Good zero-shot / few-shot capability

**Limitations / Challenges**

* Requires huge computational resources
* Risk of hallucinations (generating false / invented information)
* Latency during inference for large models
* Alignment, safety, bias, misuse are concerns

## **3. Codex (or code-generative GPT variants)**

Codex is a variant or descendant of GPT, specialized for generating code.

### **What is Codex?**

* OpenAI’s Codex is a model trained to generate programming code (Python, JavaScript, etc.) from natural language prompts.
* It’s (some version of) GPT fine-tuned on large code repositories (e.g. GitHub) to understand programming languages and generate code.

### **How it works / algorithm**

* The base architecture is still transformer (decoder-only), same as GPT.
* But the training data includes large amounts of code (source files, programming documentation, examples).
* It learns mappings from natural language instructions to code.
* During generation, you input a prompt in natural language (or partial code), and the model continues or completes code.

### **Use cases**

* Code completion (IDE integration)
* Converting natural language descriptions to functions
* Autogenerating boilerplate or helper code
* Assisting developers
* Automated code review, explanation

### **Strengths / limitations**

**Strengths**

* Great for speeding up coding / prototyping
* Understands context, can incorporate libraries, API calls

**Limitations**

* May generate code that is syntactically valid but logically incorrect
* Security, vulnerability, correctness issues
* Requires careful oversight, testing

## **4. GAN (Generative Adversarial Network)**

Switching now to generative models in vision / image domains, GANs have been very influential.

### **What is a GAN?**

* GAN = **Generative Adversarial Network**, introduced by Goodfellow et al. in 2014.
* It’s a two-player game between a **Generator** and a **Discriminator**.
* The Generator tries to generate fake data (e.g. images) indistinguishable from real data; the Discriminator tries to distinguish real vs generated (fake) data.
* They are trained together adversarially: the generator improves to fool the discriminator, the discriminator improves to detect fakes.

### **How GANs work (algorithm)**

* **Generator**: neural network that maps random noise (latent vector) z ~ p(z) to data space (e.g. images).
* **Discriminator**: neural network that takes an input (real or fake), outputs a probability that it is real.
* Training objective:  
     - Discriminator tries to maximize:  
      E[log D(x\_real)] + E[log (1 – D(G(z)))]  
     - Generator tries to minimize (or maximize negative):  
      E[log (1 – D(G(z)))]  
   (or variants like minimizing -E[log D(G(z))])
* In practice, training is delicate; often use tricks (label smoothing, gradient penalties, Wasserstein GAN, etc.)

### **Building / implementation and architectural variants**

* The networks (generator/discriminator) are often convolutional (CNNs) for images.
* Many improved variants: DCGAN, WGAN, StyleGAN (for high-quality images, controllable latent space), conditional GANs (where generation is conditioned on class labels, text), etc.
* Training stability is a big concern (mode collapse, vanishing gradients, training oscillations).

### **Use cases of GANs**

* Image synthesis (faces, objects)
* Image-to-image translation (e.g. turning sketches into images)
* Style transfer
* Super-resolution
* Data augmentation
* Generative art

### **Strengths / limitations**

**Strengths**

* Can produce very high-quality, realistic images
* Once trained, generation is fast (one forward pass)

**Limitations / challenges**

* Training is unstable
* Mode collapse (generator only produces limited variety)
* Need careful architecture & hyperparameter tuning
* Hard to control output in fine-grained ways

## **5. Diffusion models (and Stable Diffusion in particular)**

Diffusion models are a newer class of generative models (especially for images) that have outperformed GANs in many respects.

### **What is a Diffusion Model?**

* A diffusion model is a generative model that **gradually transforms noise into data** via a stochastic process (or reverse process).
* The idea: start from random noise and iteratively denoise it, guided by a learned denoising network.
* The training involves a forward diffusion (adding noise) and learning to invert (denoise).

Stable Diffusion is a specific diffusion-based model optimized for image generation (latent diffusion).

### **How they work (algorithmic principles)**

#### **Forward (noising) process**

* You take a data sample x₀ (e.g. image), then add Gaussian noise step by step to eventually reach (nearly) pure noise x\_T.
* Mathematically, you define a sequence: x\_t = √α\_t x\_{t-1} + √(1–α\_t) ε, where ε is Gaussian noise.
* Over many steps, the signal is “diffused” into noise.

#### **Reverse (denoising / generation) process**

* You train a neural network to predict the added noise (or x\_{t-1}) given x\_t and t.
* Then, at inference, you start from noise x\_T and iteratively apply the learned denoising to move toward x₀ (a plausible sample).
* This is often done via **score matching** or **denoising score matching** losses.
* More advanced versions condition on something (e.g. text prompts) so that the generated image matches a prompt — the denoiser becomes conditional. Cross-attention is useful here.

#### **Latent diffusion (Stable Diffusion)**

* Instead of working in raw pixel space, Stable Diffusion operates in a lower-dimensional latent space (learned via an autoencoder).
* The model denoises latents, then decodes to the image space. This makes computation more efficient.
* It uses a U-Net architecture as the noise-predictor, with cross-attention to the text embedding to align image with prompt.

### **Building / implementation**

* You need datasets of images.
* Pretrain an autoencoder / variational autoencoder (VAE) to map images ⇄ latents.
* Train the diffusion model (denoising network, often U-Net) on the latent space.
* Use conditioning mechanisms (text encoder, cross-attention) to guide generation.
* Use optimized sampling / inference techniques to speed up generation (e.g. fewer steps, better solvers).

### **Use cases of diffusion / stable diffusion**

* Text-to-image generation (Stable Diffusion, DALL·E 2, Midjourney, etc.)
* Image inpainting, editing, variation
* Style transfer
* Super-resolution
* Video generation (diffusion-based)
* More controllable generative tasks (because you have a stepwise generative process)

### **Strengths / limitations**

**Strengths**

* Very high image quality, often surpassing GANs in detail and diversity
* More stable training and fewer collapse issues
* Better control over generation via conditioning and intermediate steps

**Limitations / challenges**

* Slower generation (multiple steps)
* Requires many denoising steps (though improvements reduce steps)
* Computationally expensive
* Risk of artifacts if conditioning is weak or prompt poorly matched

## **6. DALL·E**

DALL·E is a model (by OpenAI) that generates images from textual prompts. It mixes ideas from GPT (transformers) and diffusion (for newer versions).

### **What is DALL·E?**

* DALL·E is a **text-to-image generation** system.
* The original DALL·E (2021) used a discrete VAE + autoregressive transformer over image tokens.
* More recent versions (e.g. DALL·E 2/3) combine transformers and diffusion models: e.g. diffusion models conditioned on text embeddings. (In broad sense, modern illustration systems follow diffusion + transformer or diffusion with text-conditioning)

### **How it works (architecture / algorithm)**

Original DALL·E (v1):

* Use a **discrete VAE** that converts images into a sequence of discrete tokens (like a codebook).
* Then, concatenate a text token sequence (prompt) + image token sequence, and train an autoregressive transformer (decoder-only) to predict image tokens after prompt tokens.
* Because images are tokenized, the transformer can treat them analogously to text tokens.
* At inference: you feed the text prompt, and the model autoregressively generates image tokens, which are decoded to pixels via VAE.

Later DALL·E versions / analogues:

* Use diffusion-based models to generate images more flexibly and with higher fidelity (e.g. image diffusion models guided by CLIP or other text embeddings).
* The process: encode prompt to embedding, then run diffusion model to generate or refine images conditioned on that embedding.

### **Use cases**

* Generative art
* Creative design, illustration
* Product concept visualization
* Advertising, gaming, storytelling
* Assistive tools (designer aids)

### **Strengths / limitations**

**Strengths**

* Create high-quality images from text
* Capable of diverse, imaginative outputs

**Limitations / challenges**

* May misinterpret prompts
* Limited control over fine details
* Bias, copyright and ethics issues
* Computational cost

|  | **Aspect** | **Diffusion Models** | **Stable Diffusion** | **GANs** | **Transformers** |
| --- | --- | --- | --- | --- | --- |
|  | **Generation** | Iterative denoising from noise | Iterative denoising in latent space | Single forward pass (generator only) | Autoregressive or parallel token generation |
|  | **Speed** | Slow (many steps) | Faster than diffusion, still iterative | Fast | Medium (depends on task) |
|  | **Architecture** | Denoising network (e.g. U-Net) | Autoencoder + denoiser U-Net | Generator + Discriminator (adversarial) | Attention-based encoder/decoder stacks |
|  | **Training Stability** | Stable, easier to train | Stable, depends on autoencoder quality | Often unstable, mode collapse issues | Generally stable |
|  | **Use Cases** | Image, audio, video generation | Efficient image generation | Image generation, style transfer | Text, code, multimodal generation |
|  | **Control** | Conditionable via embeddings | Text-conditioning via cross-attention | Conditional GANs possible but tricky | Natural conditioning with prompts |
|  | **Interpretability** | Intermediate denoising steps aid understanding | Same as diffusion | Difficult to interpret training | Attention weights provide insights |