Appendix to the book “The Essence of Algorithms” by Dmitry Popov.

Content

Neural networks and artificial intelligence

“The essence of a theoretical model is that it is a system with known properties, readily amenable to analysis, which is hypothesized to embody the essential features of a system with unknown or ambiguous properties”
(Frank Rosenblatt, “Principles of neurodynamics; perceptrons and the theory of brain mechanisms”)

Multilayer neural networks are studied within the scope of Deep Learning, which is a subset of Machine Learning. Machine Learning, in turn, is a subset of the field of Artificial Intelligence (AI).

Learning is the process of acquiring knowledge and experience.

Supervised learning is a process in which a model is trained on a labeled dataset, meaning that each training example is paired with an output label (the correct answer).

Unsupervised learning is the process where a model learns patterns from unlabeled data — that is, the data has no predefined categories or correct answers.

Instead of being told what the correct output should be, the model tries to discover the underlying structure on its own. For example, it may group similar data points together (clustering) or reduce data to its most important features (dimensionality reduction).

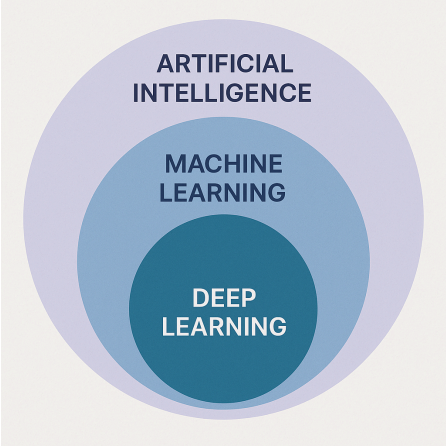
It is evident that humans are born with the ability to learn without a teacher. The human brain subconsciously classifies incoming information. Instincts drive a person to learn and explore, while reflexes, pain, and pleasure function as feedback and reinforcement (reinforcement learning).

Machine learning methods include supervised learning (regression, classification), unsupervised learning (clustering — e.g., k-means, DBSCAN — and dimensionality reduction), semi-supervised learning, reinforcement learning (Q-learning, policy-based methods), ensemble methods (bagging, boosting, random forest), deep learning (convolutional neural networks, recurrent neural networks, generative adversarial networks), and graph-based machine learning (graph neural networks).

The top-down approach (Symbolic Reasoning) models the way humans solve problems using existing knowledge and logical rules. This approach involves decomposing a problem into sub-tasks that are solved sequentially based on general principles and logical reasoning.

The bottom-up approach (Neural Networks) models the structure of the human brain, which consists of a large number of simple units called neurons. Each neuron acts as a weighted average of its inputs, and we can train a neural network to solve useful tasks by providing it with training data.

Note that the domain (or set) of Artificial Intelligence is broader than that of Machine Learning. This means that a machine can exhibit intelligent behavior without the ability to learn.



According to definition of ISO/IEC TR 29119-11: “Artificial intelligence - the capability of an engineered system to acquire, process and apply knowledge and skills”.

According to the definition of the European Commission AI HLEG: "Artificial intelligence (AI) refers to systems designed by humans that, given a complex goal, act in the physical or digital world by perceiving their environment, interpreting the collected structured or unstructured data, reasoning on the knowledge derived from this data and deciding the best action(s) to take (according to predefined parameters) to achieve the given goal. AI systems can also be

designed to learn to adapt their behaviour by analysing how the environment is affected by their previous actions".

Artificial intelligence (AI) refers to systems that display intelligent behaviour by analysing their environment and taking actions – with some degree of autonomy – to achieve specific goals. AI-based systems can be purely software-based, acting in the virtual world (e.g. voice assistants, image analysis software, search engines, speech and face recognition systems) or AI can be embedded in hardware devices (e.g. advanced robots, autonomous cars, drones or Internet of Things applications).

Note that passing the well-known Turing Test is not a necessary condition for calling a program artificial intelligence. In principle, artificial intelligence falls under the definition given by physicist Michio Kaku in his book "The Future of the Mind": "Consciousness is the process of creating a model of the world using multiple feedback loops in various parameters (e.g., in temperature, space, time, and in relation to others), in order to accomplish a goal (e.g., find mates, food, shelter). For example, the lowest level of consciousness is Level 0, where an organism is stationary or has limited mobility and creates a model of its place using feedback loops in a few parameters (e.g., temperature). For example, the simplest level of consciousness is a thermostat. It automatically turns on an air conditioner or heater to adjust the temperature in a room, without any help. The key is a feedback loop that turns on a switch if the temperature gets too hot or cold. (For example, metals expand when heated, so a thermostat can turn on a switch if a metal strip expands beyond a certain point.)".

If you disagree with this definition of Artificial Intelligence, read about the AI Effect.

The AI Effect is a cognitive bias in which people stop considering a certain technology as artificial intelligence once it starts working reliably and becomes commonplace. In other words, as soon as a task is automated and perceived as "simple," it gets excluded from the concept of AI.

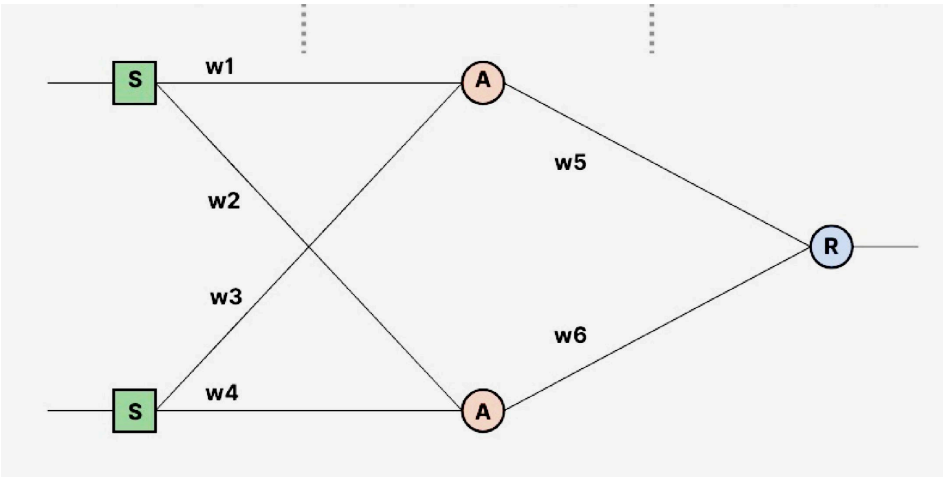
Feedforward neural networks (FNN) are historically the first and simplest type of neural networks. They were first described by Frank Rosenblatt under the name Perceptron in his book published in 1957. A feedforward neural network (FNN) has no loops, unlike recurrent neural networks (RNN), which do have cycles.

Feedforward neural networks can be used for various tasks, including modeling logical operators like XOR, classifying emails as spam or not spam, creating image filters, making different types of predictions and recommendations—such as movie suggestions—or even choosing text color based on the background.

Frank Rosenblatt also proved that a multilayer perceptron (FNN) can implement the XOR operator and solve any classification task with respect to its inputs.

With an FNN (Feedforward Neural Network), it is even possible to recognize letters. However, for more complex pattern recognition tasks, Convolutional Neural Networks (CNN), proposed by Yann LeCun, are more suitable.

We will represent a feedforward neural network (FNN) as a graph. The nodes (vertices) of the graph represent artificial neurons, analogous to natural neurons in the brain.
A neuron receives input signals, sums them up, and activates according to an activation function—that is, it sends a signal to the output. We do not take into account the lengths of the connections (edges) in the graph and assume that signals propagate in the same amount of time through all paths.



The diagram depicts a Feedforward Neural Network (FNN), which is capable of learning to approximate binary Boolean operators such as XOR.

Imagine that each connection w_i has a certain electrical resistance, and thus the signal passing through it will have a strength corresponding to that resistance. The core idea of this neural network is that these resistances (or weights) dynamically change during the training process. We input signals into the input layer (elements S) and receive the result at the output layer (element R). If the result is satisfactory, we increase the weights; if not, we decrease the weights of specific connections.

Through this feedback and reinforcement mechanism, the network is trained. Before training, the weights (resistance in physical terms) can be initialized randomly. If the network has hundreds of nodes (neurons), the programmer does not know the weight distribution after training—more precisely, they can view or save it, but in general, they don't need to know it unless they want to configure another network similarly. Thus, the configuration of an FNN consists of its structure and weight matrix.

An artificial neuron has an activation function—i.e., it sends a signal to the output if the sum of its inputs, passed through the activation function, activates it. Some nodes may not have an activation function, for example, the input elements (S in the diagram).

For activation functions, the following are used in different cases:

Heaviside discontinuous function := x≥0 ? 1 : 0;

Sigmoid (σ) = $e^x/(e^x+1) = 1/(1+e^{-x})$,

ReLU = max(0, x), and others.

Greater preference is given to "smooth functions" rather than discontinuous ones because smooth functions are continuously differentiable, which allows for the use of the gradient descent algorithm to correct inaccuracies in the results.

Sigmoid derivative σ'(x) = σ(x) * (1 — σ(x));

Let's remove cognitive dissonance and shift from describing a neural network as a physical structure with wires, resistance, electricity, and transistors to describing it as a set of functions and matrices.

A neural network, in mathematical terms, is a set of functions. Each layer of the network is a function that takes an array of arguments and returns an array of arguments for the next layer.

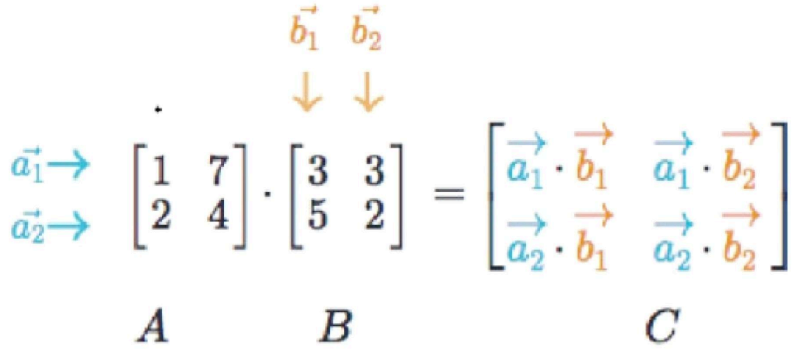
We will later replace the term array with vector. Each such layer function contains functions representing individual neurons (and their activation functions) and a matrix of the weights of their connections. In Object-Oriented Programming (OOP), layer objects are created using classes.

A matrix is a table of numbers. A matrix with one row or one column is called a vector.
For two matrices A and B to be multiplied, the number of columns in matrix A must equal the number of rows in matrix B, or vice versa.

Matrix multiplication can be defined based on the scalar product of vectors.

a = <x,y>;
b = <v,w>;
a * **b** = xv + yw; (scalar product).

Matrix multiplication



Now let's calculate the network shown in the diagram above.

Input layer

A =

1	0

B =

w_1	w_2
w_3	w_4

A * **B** =

$1 * w_1 + 0 * w_3$	$1 * w_2 + 0 * w_4$

Hidden layer

C =

sigmoid(1 * w_1 + 0 * w_3)	sigmoid(1 * w_2 + 0 * w_4)

C *

w_5
w_6

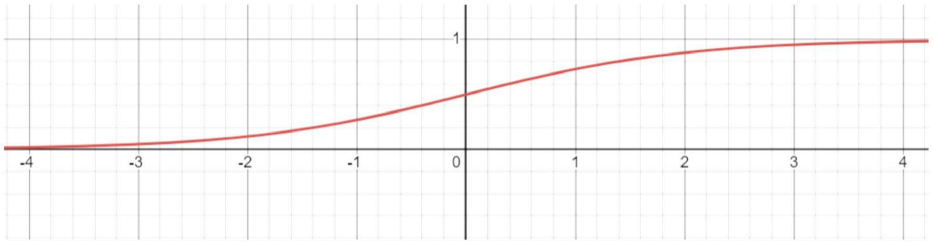
= sigmoid(sigmoid(1 * w_1 + 0 * w_3) * w_5 + sigmoid(1 * w_2 + 0 * w_4) * w_6).

We sum these values and pass them through the activation function (sigmoid — the logistic function).

Bias in feedforward neural networks (FNN) is an additional parameter added to the weighted sum of input values before they are passed through the activation function of the neurons in each layer of the network. Adding bias allows the activations of the neurons to start from a value other than zero.

By the way, the Strassen algorithm can be used for matrix multiplication. In general, multiplying large matrices is a complex algorithm, so for optimization, fast programming languages and GPU computations are used.

Sigmoid graph



Mean squared error (MSE) is used for the loss function.
Simply, MSE can be written as: (prediction – actualValue)².

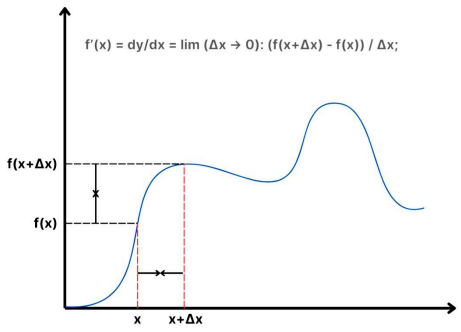
Raising to the power is necessary to make the function smooth, eliminate negative values, and ensure that larger deviations contribute more to the sum.
After that, backpropagation and gradient descent are performed.
Backpropagation computes the gradient of the loss function with respect to the network weights for a single input-output example, and does so efficiently by calculating the gradient of one layer at a time, iterating backward from the last layer. Gradient descent or its variants, such as stochastic gradient descent, are commonly used.
Each layer can be represented as a scalar function of many variables.
We can use the vector operator nabla (∇) to determine the gradient of this function, essentially a set of partial derivatives of the function. Once we know these derivatives, we can update the weights of a given layer, then move to the previous layer and also determine the partial derivatives, adjusting the parameters to match the results of the next layer.

The gradient of a scalar function f is written as:
$\nabla f = (\partial f/\partial x, \partial f/\partial y, \partial f/\partial z)$.

Derivative of a function of a real variable

Let f be a real function defined on the interval [a, b].

Taking an arbitrary number $x \in [a, b]$, we define:
$f'(x) = \lim(t \to x) ((f(t) - f(x)) / (t - x))$, where $a<t<b$, $t \neq x$.
Also:
$f(t) - f(x) = dy$;
$t - x = dx$;
$f'(x) = dy/dx = \lim(t \to x) (dy/dx)$;

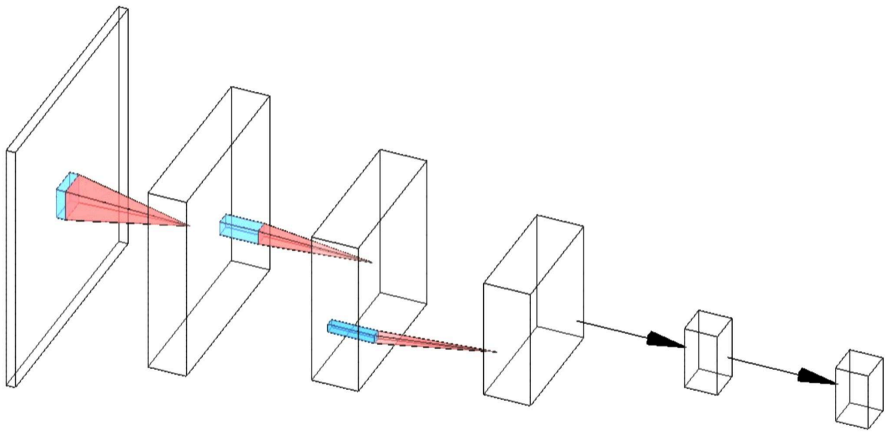


FNN JS example

```
// Функція активації - сигмоїда
function sigmoid(x) {
  return 1 / (1 + Math.exp(-x));
}
// Похідна функції активації
function sigmoidDerivative(x) {
  return x * (1 - x);
}
class NeuralNetwork {
  constructor(inputNodes, hiddenNodes, outputNodes) {
    this.inputNodes = inputNodes;
    this.hiddenNodes = hiddenNodes;
    this.outputNodes = outputNodes;
    // Випадкова ініціалізація ваг
    this.weightsInputHidden = Array(this.hiddenNodes)
      .fill()
      .map(() => Array(this.inputNodes).fill().map(() => Math.random() - 0.5));
    this.weightsHiddenOutput = Array(this.outputNodes)
      .fill()
      .map(() => Array(this.hiddenNodes).fill().map(() => Math.random() - 0.5));
    // Зсуви
    this.biasHidden = Array(this.hiddenNodes).fill().map(() => Math.random() - 0.5);
    this.biasOutput = Array(this.outputNodes).fill().map(() => Math.random() - 0.5);
  }
  // Метод навчання
  train(inputs, targets, learningRate) {
    // --- Feedforward ---
    // Вхідні дані -> Прихований шар
    const hiddenInputs = this.weightsInputHidden.map(weights =>
      weights.reduce((sum, weight, i) => sum + weight * inputs[i], 0)
    );
    const hiddenOutputs = hiddenInputs.map(sigmoid);

    // Прихований шар -> Вихідний шар
    const finalInputs = this.weightsHiddenOutput.map(weights =>
      weights.reduce((sum, weight, i) => sum + weight * hiddenOutputs[i], 0)
    );
    const finalOutputs = finalInputs.map(sigmoid);
    // --- Backpropagation ---
    // Похибки на виході
    const outputErrors = finalOutputs.map((output, i) => targets[i] - output);
    // Похідна функції активації для вихідного шару
    const outputGradients = finalOutputs.map(sigmoidDerivative);
    // Зміна ваг між прихованим і вихідним шарами
    const weightsHiddenOutputDeltas = outputErrors.map((error, i) =>
      hiddenOutputs.map(output => output * error * outputGradients[i] * learningRate)
    );
    this.weightsHiddenOutput = this.weightsHiddenOutput.map((weights, i) =>
      weights.map((weight, j) => weight + weightsHiddenOutputDeltas[i][j])
    );
    // Похибки прихованого шару
    const hiddenErrors = this.weightsHiddenOutput.reduce((errors, weights, i) =>
      errors.map((error, j) => error + weights[j] * outputErrors[i]),
      Array(this.hiddenNodes).fill(0)
    );
    // Похідна функції активації для прихованого шару
    const hiddenGradients = hiddenOutputs.map(sigmoidDerivative);
    // Зміна ваг між вхідним і прихованим шарами
    const weightsInputHiddenDeltas = hiddenErrors.map((error, i) =>
      inputs.map(input => input * error * hiddenGradients[i] * learningRate)
    );
    this.weightsInputHidden = this.weightsInputHidden.map((weights, i) =>
      weights.map((weight, j) => weight + weightsInputHiddenDeltas[i][j])
    );
  }
  // Метод передбачення
  predict(inputs) {
    // Вхідні дані -> Прихований шар
    const hiddenInputs = this.weightsInputHidden.map(weights =>
      weights.reduce((sum, weight, i) => sum + weight * inputs[i], 0)
    );
    const hiddenOutputs = hiddenInputs.map(sigmoid);
    // Прихований шар -> Вихідний шар
    const finalInputs = this.weightsHiddenOutput.map(weights =>
      weights.reduce((sum, weight, i) => sum + weight * hiddenOutputs[i], 0)
    );
    const finalOutputs = finalInputs.map(sigmoid);

    return finalOutputs;
  }
}
// Вхідні дані та очікувані виходи для функції XOR
const xorInputs = [
  [0, 0],
  [0, 1],
  [1, 0],
  [1, 1]
];
const xorTargets = [
  [0],
  [1],
  [1],
  [0]
];
const inputNodes = 2;
// Створення мережі
const nn = new NeuralNetwork(inputNodes, inputNodes + 1, 1);
// Навчання мережі
for (let i = 0; i < 100000; i++) {
  const index = Math.floor(Math.random() * xorInputs.length);
  nn.train(xorInputs[index], xorTargets[index], 0.1);
}
// Передбачення
xorInputs.forEach(input => {
  console.log(`Input: ${input} Output: ${nn.predict(input)}`);
});
```

Convolutional Neural Network

The first references to Convolutional Neural Networks (CNNs) belong to the works of Yann LeCun and his colleagues (see the Neocognitron by Fukushima). One of the most famous works of LeCun and his team describes LeNet-5, a neural network that was successfully used for handwritten digit recognition on the MNIST database.
The MNIST database (short for Mixed National Institute of Standards and Technology) contains 60,000 images for training and 10,000 images for testing.
In 2012, Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton presented the AlexNet architecture, which significantly improved results in image recognition competitions and became a catalyst for the popularization of deep learning.
In 2013, a team at Google, led by Tomas Mikolov, created word2vec, a word embedding toolkit that can train vector models faster than previous approaches.



A Convolutional Neural Network (CNN) is a special kind of neural network mainly used for analyzing visual data — like photos or videos. It's especially good at recognizing patterns, like edges, shapes, or even faces.

At a high level, a CNN works through three main ideas:

1. Convolution
Instead of looking at the whole image at once, CNN looks at small pieces.
It uses small filters (also called kernels) that slide over the image and extract important features — like lines, corners, colors, etc.
Mathematically, this is done by multiplying parts of the image with the filter and adding the results. This gives a new "filtered" image, called a feature map.

You can imagine it like using a tiny magnifying glass to examine tiny sections of a photo.

2. Pooling (Subsampling)
After convolution, CNN simplifies the data using pooling.
It takes a small area (like a 2x2 block) and picks the largest value (or sometimes the average).
This reduces the size of the data and keeps only the most important information, making the network faster and less sensitive to small shifts or noise in the image.

Think of it like zooming out — you lose tiny details, but you still see the big important features.

3. Fully Connected Layers
After several rounds of convolution and pooling, CNN "flattens" the extracted features into a long list and passes it through regular neural network layers (like in classic Machine Learning).
These layers then make a decision — for example: "Is this a cat or a dog?"

Summary
A CNN first extracts features (with convolution), then compresses information (with pooling), and finally makes a decision (with fully connected layers).

Visually:
Input Image → Convolution → Pooling → Convolution → Pooling → Fully Connected Layers → Output.

In a Convolutional Neural Network (CNN), the dimensions of the data change as it passes through each layer. Starting with a grayscale image of size 28x28x1 (height × width × channels), a convolutional layer with a 3x3 filter and no padding reduces the spatial dimensions to 26x26, while the depth increases based on the number of filters (e.g., 32 filters produce an output of 26x26x32). A pooling layer, such as 2x2 max pooling, then halves the height and width, reducing the output to 13x13x32. This process may repeat, further reducing the dimensions (e.g., another convolution may result in 11x11x64, followed by 5x5x64 after pooling). After the last pooling layer, the 3D data is flattened into a 1D vector (e.g., 5×5×64 = 1600 values) and passed to fully connected layers that lead to the final prediction.

The overall change in dimensions is governed by a few key parameters: filter size, stride (how far the filter moves), and padding (whether the input is bordered with zeros to preserve size). Using padding can help retain the original spatial dimensions after convolution (e.g., 28x28 remains 28x28), while a larger stride shrinks the output faster. Pooling always reduces the width and height, helping to downsample the data and reduce computational load while keeping the most important features. Understanding how these operations affect dimensions is crucial when designing CNN architectures, as it ensures the network structure matches the desired input and output sizes.

Input image: 28x28x1
(Width × Height × Channels)
(1 channel because it's grayscale; if it were color, there would be 3 channels: RGB)

OpenCV is a powerful open-source library for image processing and computer vision tasks like face detection and object tracking, while PyTorch and TensorFlow are popular deep learning frameworks—PyTorch favored for flexibility and research, TensorFlow for large-scale production. YOLO (You Only Look Once) is a fast, real-time object detection algorithm that detects multiple objects in a single pass, whereas Detectron2, built on PyTorch by Facebook, provides more accurate but often slower detection and segmentation tools. AWS Rekognition is a fully managed cloud service that offers pre-trained computer vision capabilities like facial analysis and object recognition via simple APIs, ideal for scalable applications without model training. face-api.js is a JavaScript library that provides facial recognition and face detection capabilities directly in the browser using HTML5's getUserMedia API. It allows developers to identify faces, detect facial landmarks (like eyes and mouth), and perform face recognition tasks on images or video streams without requiring a server-side backend. It leverages deep learning models, such as those trained in TensorFlow.js, to run directly in the browser, making it fast and suitable for real-time applications like user authentication or emotion detection.

Transformers

In 2017, Google researchers introduced the Transformer model, which revolutionized natural language processing. The Transformer uses an attention mechanism that allows the model to process large data sequences more efficiently. This led to the creation of models like BERT, GPT, and other modern language models.
In 2020, OpenAI introduced GPT-3 (Generative Pre-trained Transformer 3), one of the most powerful language models at the time, capable of generating text that is almost indistinguishable from human-written content.

When you type a sentence like **"The sun is"**, a **transformer model** tries to predict the next word (e.g., **"shining"**) by transforming your input into a deep understanding of language, meaning, and context — all through a structured pipeline of operations.

1. **Tokenization**
   First, the sentence is broken down into **tokens**, which are not necessarily full words — they could be subwords, like `"sh"`, `"ine"`, `"ing"`. This allows the model to handle rare or new words.

2. **Vector Embeddings**
   Each token is then mapped to a high-dimensional **embedding vector**, which is a numerical representation that captures the token's meaning.
   These embeddings are initially random, but during training (on massive datasets), the model adjusts them so that similar words end up close together in vector space (like in **Word2Vec**, but more dynamic and context-aware).

3. **Position Encoding**
   Since transformers don't have a built-in sense of word order (no recurrence), we **add position information** to each embedding, so the model knows the token's place in the sequence.

4. **Self-Attention**
   Now, the model passes these vectors through the **attention mechanism**, which allows every token to look at (or "attend to") every other token in the sentence.
   For example, in "The animal that chased the cat was fast," the word "was" might attend strongly to "animal" rather than "cat," because it learns that "animal was fast" is more likely.

Self-attention calculates weights (numbers between 0 and 1) for how much each token should influence every other token.

5. **Feed-Forward Neural Networks (FNN)**
   After attention refines the meaning of each token based on its context, a **feed-forward neural network** is applied independently to each token vector. This helps the model extract and transform deeper features.

6. **Stacked Layers**
   Transformers consist of **many such layers** (attention + FNN), stacked on top of each other. Each layer refines the token representations even more.

7. **Prediction of Next Word**
   Finally, once the input sentence is processed through all layers, the model outputs a **probability distribution** over its entire vocabulary — it calculates, for example:

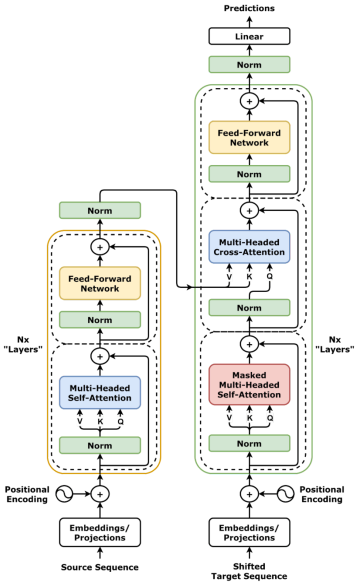
- 60% chance the next word is "shining"
- 30% chance it's "rising"
- 10% chance it's "hot"

The word with the highest probability is chosen as the output — that's how the transformer **predicts the next word**.

In essence, a transformer is a deep stack of mathematical functions that:

- Turns raw text into numbers (via tokens and embeddings),

- Mixes them using attention (so each word "understands" its neighbors),
- Refines them using neural networks,
- And finally produces meaningful predictions based on everything it has seen during training.



(dvgodoy • CC BY 4.0)

Word2Vec is a technique used to convert words into numerical vectors, where the vector of a word captures its meaning and context. The main idea is to train a model that can predict which words are likely to appear around a given target word. This is done by using a neural network that takes a word as input and learns to predict the surrounding words in a sentence. For example, given the word “cat,” the model might predict that words like “pet,” “animal,” and “fur” are likely to appear nearby. This is done within a specific "window size" that defines how many words before and after the target word are considered as context.
There are two main types of Word2Vec models: Skip-Gram and Continuous Bag-of-Words (CBOW). In the Skip-Gram model, the goal is to predict the context words (words around the target word) given a central word. In the CBOW model, the task is the opposite: given a set of surrounding words (context), the model predicts the central word. Both models work by

converting words into high-dimensional vectors, which are learned and refined during training. These vectors, or "embeddings," capture the semantic meaning of the words, so that similar words end up with similar vector representations.
To achieve this, Word2Vec uses a weight matrix that acts as a "lookup table" for the word embeddings. Initially, the values in the matrix are random, but as the model trains, it adjusts the weights to improve the prediction accuracy. When the model makes a prediction, it outputs a vector that corresponds to a word in the vocabulary, and a softmax function is used to turn this output into a probability. The training process tries to minimize the error in these predictions, updating the weights over time. The dimensionality of the word vectors (usually between 100 and 1,000) affects how well the model captures relationships between words, with larger dimensions generally providing more detailed embeddings, but with diminishing returns after a certain point.
In short, Word2Vec creates dense numerical representations of words that encode their meanings, allowing the model to understand word relationships based on their context. This is powerful for a range of NLP tasks, like search, translation, and semantic analysis.

A one-hot vector is a way of representing a word (or any categorical data) as a binary vector where all elements are 0 except for one element, which is set to 1. It's a simple but common method to encode discrete items, like words, in a format that can be processed by a machine learning model.

Example:

Imagine you have a vocabulary of 5 words:

"cat",
"dog",
"fish",
"bird",
"lizard",

A one-hot encoding for each of these words would look like this:

"cat" → [1, 0, 0, 0, 0],
"dog" → [0, 1, 0, 0, 0],
"fish" → [0, 0, 1, 0, 0],
"bird" → [0, 0, 0, 1, 0],
"lizard" → [0, 0, 0, 0, 1]

Each word is represented as a vector of length 5 (the number of words in the vocabulary), where the position of the 1 corresponds to the word’s index in the vocabulary, and all other positions are set to 0.

In the context of Word2Vec:

One-hot vectors are used to represent the words in the input data when training the model. For example, if the word "cat" is the input, its one-hot vector would look like [1, 0, 0, 0, 0], where only the "cat" position is 1, and all other positions are 0.
The input one-hot vector is then multiplied by a weight matrix (usually of size vocabulary x embedding dimension) to get a dense vector (embedding) that represents the word's meaning in a high-dimensional space.

After training, the model will replace one-hot vectors with word embeddings (dense vectors that capture semantic meaning).

The downside is that one-hot vectors are sparse (most elements are 0) and they don't capture any relationships between words. For example, the vector for "cat" doesn't tell you anything about how it might be related to "dog." That's where word embeddings come in—they transform one-hot vectors into dense vectors that can capture these relationships.

The king and queen example is often used to illustrate how word embeddings, like those learned by Word2Vec, capture relationships between words. This example demonstrates how vector operations on word embeddings can reveal semantic relationships, such as gender or royalty.

Here's how it works:

1. Word Embeddings:
After training a Word2Vec model on a large corpus of text, the model learns to represent words as dense vectors (embeddings) in a high-dimensional space. These embeddings capture the meanings of words, as well as their relationships to other words.

2. Example: King - Man + Woman = Queen:
Word embeddings can capture complex relationships such as analogies. Here's how you might think about it mathematically using vector operations:

King and Queen are related words, both being royalty, but King is male and Queen is female.
Man and Woman represent gender differences.

In the vector space, the relationship between King and Queen is analogous to the relationship between Man and Woman. More formally:

King - Man + Woman = Queen

3. How It Works:

When the model learns the vector for King, it understands not only that it’s a royal figure, but also that it has certain attributes (like being male).

Similarly, Queen is learned as a royal figure but with the female attribute.

Man and Woman are learned as related words, with Man representing male and Woman representing female.

When you subtract Man from King, you’re essentially removing the "male" part of the vector, leaving just the "royalty" part.

Then, adding Woman adds the "female" part to the vector, resulting in the vector for Queen.

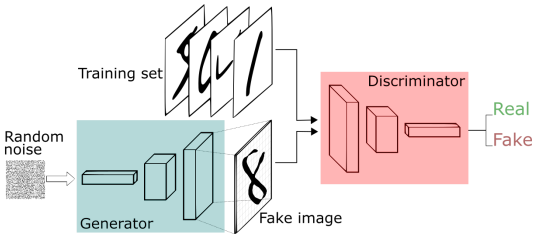
4. Geometrically:

In the vector space, the relationship between words is preserved geometrically. So, when you subtract one vector from another (e.g., King - Man), you’re essentially removing the dimension of meaning that corresponds to the "man-ness" of the word. Then, adding another vector (e.g., Woman) shifts the result in the direction of "woman-ness." The resulting vector is closest to the Queen vector.

As of 2022, the straight Word2vec approach was described as "dated". Transformer-based models, such as ELMo and BERT, which add multiple neural-network attention layers on top of a word embedding model similar to Word2vec, have come to be regarded as the state of the art in NLP.

Generative Adversarial Networks

Generative Adversarial Networks, or GANs, are a type of machine learning model invented in 2014 by Ian Goodfellow and his collaborators. They consist of two neural networks that compete with each other in a process that resembles a game. One network, called the generator, creates data intended to mimic real data—for example, it might try to generate images that look like photographs of human faces. The other network, called the discriminator, evaluates data and tries to determine whether a given input is real (from the actual dataset) or fake (created by the generator). These two networks are trained together: the generator improves by learning to fool the discriminator, and the discriminator improves by getting better at detecting fakes. Over time, this adversarial process pushes the generator to produce increasingly realistic outputs. GANs have become a powerful tool in many areas, including image generation, style transfer, super-resolution, and even the creation of synthetic data for training other models. Their ability to generate high-quality, lifelike content has also led to their use in applications such as deepfakes and artistic content creation.



(Image credit: Thalles Silva, https://www.freecodecamp.org)

Generator and Discriminator—interact during training:

1. Random Noise Input

The process begins with random noise fed into the Generator. This noise is just a vector of numbers with no structure, like static.

2. Generator (G)

The Generator transforms this noise into a fake sample, for example, an image that tries to mimic the style of the real training data (like a face photo).
Its goal is to fool the Discriminator into thinking this output is real.

3. Real Data vs Fake Data

The Discriminator (D) receives two types of input:
a) Real data from the actual dataset (e.g., real photos)
b) Fake data generated by the Generator

4. Discriminator (D)

The Discriminator is trained to distinguish between real and fake data.
It outputs a probability score: high for real data, low for fake data.

5. Feedback Loop

The Discriminator's feedback helps both networks improve:
a) It learns to better detect fakes.
b) The Generator uses that feedback to produce more convincing fakes in the next round.

6. Training Iteration

This loop continues: the Generator keeps getting better at generating realistic data, and the Discriminator keeps getting better at spotting fakes—until the fake data becomes indistinguishable from real data.

DIKW pyramid

The DIKW pyramid is a model that illustrates how raw data (D) transforms into information (I) through the addition of context, then into knowledge (K) through understanding, and ultimately into wisdom (W) — the ability to make the right decisions based on knowledge.

Data: Raw facts and figures without context.
Information: Processed data with meaning.
Knowledge: Information that has been analyzed for understanding.
Wisdom: The application of knowledge with insight and judgment.

Human intuition is a matter of habit, and in this, David Hume was right. Indeed, most of what we call physical intuition, manners, even language, is a matter of habit — more precisely, training based on many examples, like in deep learning.
But David Hume went too far in many ways.
Likewise, it is worth saying that a person is not a tabula rasa — even newborns have a basic ability to recognize faces from birth.
Humans, as noted well in the Book of Genesis, have the ability to give names — that is, to classify objects.

Humans have an innate ability to count using natural numbers. This relates to Kronecker's famous phrase about natural numbers ("God made the natural numbers; all else is the work of man")
What did Descartes mean by the notion of thinking? A being that perceives, remembers, thinks, analyzes.
Regarding memory: yes, a person can remember — but what happens if you replace the memories of an adult? There will be a conflict between memories, perceptions, and a person's principles. Because a person is shaped by experience. If experience is replaced, it will not correspond to the present person. This is what Hegel referred to as the development of the spirit.
Human consciousness is single-threaded. Experiments have shown that when focusing on a specific task, a person may not notice other details.
Also, people have the ability to predict based on common behavior and experience.
The brain has many different mechanisms — for example, binocular vision, fliping the image formed on the retina. That is, light passes through the pupil and forms an image on the retina that is inverted both vertically and horizontally.
The brain also generates various colors using just three types of color receptors — cones and rods in the retina.
The brain follows precise rules to perceive some sounds as pleasant and others as unpleasant.
A person has six senses: sight, hearing, smell, taste, touch, and balance. And five tastes: saltiness, sweetness, bitterness, sourness, and savoriness.
Human perception follows the Weber–Fechner law.
Miller's Law, proposed by psychologist George A. Miller in 1956, states that the average number of items an adult can hold in short-term memory is 7 ± 2. This means most people can remember about 5 to 9 chunks of information at once.
A chunk can be a digit, letter, word, or meaningful group of items (e.g., "CIA-FBI" as two chunks rather than six letters).
Memory can be impaired by alcohol, for example, and alcohol can cause amnesia.
The brain has different layers and consists of neurons, which are controlled by neurotransmitters.
It is now evident that the human brain contains a neural network(s), which is partially modeled by powerful artificial neural networks built on artificial neurons.
More interestingly, people move from example-based thinking, such as recognition, to deductive thinking based on certain mathematical rules.
Kant suggests that mathematics is based on human perception, so it's deeply human. But why does it work? One answer comes from the theory of evolution: we would be dead if it didn't work.

Like a convolutional neural network (CNN), the brain does not perceive complex objects as a whole and all at once. Instead, it analyzes them step by step: breaking them down into individual features, fragments, and details. This fact was already known to Augustine of Hippo when he asked us to imagine a 1000-sided polygon. It is also illustrated by the Penrose triangle.

When perceiving new objects or situations, the brain compares them with examples from previous experience. This allows it to recognize familiar patterns faster and more accurately — an effect similar to how a convolutional neural network, once trained, can classify images with high confidence.

Neural networks can also be recurrent and context-dependent.

If you've been looking at flowers for a while, your brain activates associated images, shapes, colors, and associations. Later, when shown ambiguous or abstract images (such as Rorschach inkblots), your brain is inclined to interpret them based on this recently activated experience.

Priming is a psychological effect where previous experience or perception influences your subsequent behavior, thoughts, or perception — even if you're not consciously aware of it.

For example, if someone is shown the word "yellow" and then asked what word comes to mind when they hear "fruit," they are more likely to say "banana" than "apple," because yellow and banana are associated.

During thinking, various ideas, hypotheses, and intuitive guesses arise. But not all of them are immediately accepted. Many go through an internal verification — a critical evaluation process that resembles the role of the discriminator in a Generative Adversarial Network (GAN): filtering out false, illogical, or implausible options.

In addition, your brain calculates the most probable outcomes or actions, similar to how LLMs or Bayesian networks operate.

Sources:

"Штучні нейронні мережі: базові положення", І.А. Терейковський, Д.А. Бушуєв (КПІ),

"Штучні нейронні мережі: обчислення", М.А. Новотарський, Б.Б. Нестеренко (НАНУ),

"Learning TensorFlow.js: Powerful Machine Learning in JavaScript" by Gant Laborde,

"Deep Learning with JavaScript" by Eric D. Nielsen, Shanqing Cai, and Stanley Bileschi,

"The Hundred-page Machine Learning Book" by Andriy Burkov,

"Deep Learning: A Practitioner's Approach" by Josh Patterson, Adam Gibson,

"Deep learning" by Ian Goodfellow, Yoshua Bengio, Aaron Courville,

"Foundations of Statistical Natural Language Processing" by Christopher D. Manning,

“Pattern Recognition and Machine Learning” by Christopher Bishop,

“Certified Tester AI Testing Syllabus” by ISTQB,

"Attention Is All You Need" by Ashish Vaswani, Aidan Gomez,

“Deep Learning Basics: Introduction and Overview” by Lex Fridman (on YouTube),

“MIT Introduction to Deep Learning” by Alexander Amini (on YouTube),

“Lex Fridman - Yann LeCun, Podcast #36” (on YouTube).

“Encyclopedia of Artificial Intelligence” by Juan Ramón, Julián Dorado.

“Encyclopedia of Machine Learning”, by Claude Sammut, Geoffrey I. Webb (Eds.).

Quantum computing

A universal quantum computer can be considered an extension of the classical computer (Turing machine) by adding the ability to work with quantum systems using quantum operators.
A qubit is an extension of a bit. A bit can have two states—1 or 0, meaning on/off. A qubit also has two states after measurement, but before measurement, it can exist in either of these states or in their superposition—meaning it has a certain probability of being in one of the two basic states after measurement.
For example, if a qubit has a 30% probability of being in state 1 after measurement, conducting a large number of measurements will give approximately 70% results in state 0 and 30% in state 1.
A qubit is described as a superposition of basis states with certain complex amplitudes. The probability of obtaining state 1 after measurement equals the square of the magnitude of the amplitude of state 1.
The qubit itself can be represented by a photon, electron, or ion. All of these are quantum systems.
Superposition manifests in experiments such as the double-slit experiment, Stern-Gerlach experiments, and observations of radioactive decay, among others.
However, when working with a single qubit, you only operate with two states and their superposition. It becomes more interesting when multiple qubits are used. When multiple quantum systems—qubits—interact, quantum entanglement comes into play.
For instance, if one qubit can simultaneously be in two states, two qubits can be in four states at once.

16 qubits simultaneously exist in a superposition of 65,536 basic states (all possible combinations of zeros and ones). However, during measurement, we will only obtain one of these numerous states.
The essence of quantum algorithms lies in using quantum gates to perform operations on qubits in a superposition state, such that when measured, the result with a higher probability is the one that answers the problem at hand.
There are many quantum algorithms, among which the Deutsch algorithm, Shor's algorithm, and Grover's algorithm are well-known.
Quantum gates implement reversible computations, particularly using the Toffoli gate. This means each operation can be undone.
Other quantum gates (Hadamard, CNOT, Pauli-X, etc.) are also used in quantum computing, and they are unitary.
Quantum algorithms are probabilistic, not deterministic.

In classical wave theory, the amplitude of a wave is the quantity that describes the maximum deviation of a particle or field from equilibrium. In the case of quantum systems, the amplitudes for each possible state (such as $|0\rangle$ and $|1\rangle$) also have a wave-like nature. Quantum mechanics uses wave functions that describe the probabilities of different measurement outcomes. That is, a qubit in superposition becomes like a wave, encompassing all possible states, with different amplitude values for each.
For example, when a qubit is in a superposition of states, its wave function may be a combination of waves that have different amplitudes and phases.
Quantum mechanics uses wave interference as a mechanism to achieve specific results. Just as classical waves can amplify or diminish one another, quantum states can combine through amplitude interference. When the amplitudes for different states of a qubit have the same phase, the probability of that state increases. If the amplitudes have opposite phases, the probability of that state may decrease (or cancel out).
This phenomenon is the basis of quantum algorithms, where quantum states are manipulated in such a way as to enhance the probability of obtaining the correct result through wave interference.

Each qubit can be represented as a superposition of two basis states ($|0\rangle$ and $|1\rangle$), where each state has its own amplitude, denoted as α for $|0\rangle$ and β for $|1\rangle$:

$$\psi = \alpha|0\rangle + \beta|1\rangle$$

Here, α and β are complex numbers that represent the amplitudes of the $|0\rangle$ and $|1\rangle$ states. The probability of the qubit being measured in the $|0\rangle$ state is equal to the square of the modulus of amplitude α (i.e., $|\alpha|^2$), and the probability of it being found in the $|1\rangle$ state is the square of the modulus of β (i.e., $|\beta|^2$).

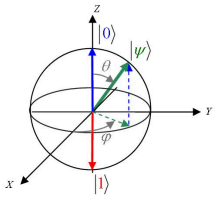
The sum of the probabilities must equal 1:

$$|\alpha|^2 + |\beta|^2 = 1$$

This means that when you measure the state of a qubit, the probability of each possible outcome (0 or 1) is determined by these amplitudes.

Thus, a superposition is a state in which the qubit "exists" in both basis states at the same time (with different probabilities), but after measurement, you get a definite result.

Bloch sphere



The Bloch sphere is a geometric representation of qubit states (the fundamental unit of quantum information) as points on or inside a unit sphere.
It provides an intuitive way to visualize how quantum operations (gates) act — they simply rotate the qubit around the sphere's axes.
If a point lies on the equator of the Bloch sphere, the qubit has an equal probability of being 0 or 1, but with a specific phase (a rotation in space).

The No-Cloning Theorem
The no-cloning theorem is a fundamental principle of quantum mechanics which states:

“It is impossible to create an exact copy of an arbitrary unknown quantum state”.

Essence

Suppose we have a quantum object in the state |ψ⟩, and we want to copy it. In the classical world, this is trivial — copying a file, text, or number. But in quantum mechanics, due to the linearity and unitarity of state evolution, this is physically impossible.

Formal Statement

Assume there exists a universal quantum operation (a unitary operator) U that can clone any state:

$$U|\psi\rangle|e\rangle = |\psi\rangle|\psi\rangle,$$

where $|\psi\rangle$ is any arbitrary state, and $|e\rangle$ is the "blank carrier" state (e.g., a qubit ready to be written).

If this works for both $|\psi\rangle$ and $|\varphi\rangle$, then the following must hold:

$$\langle\psi|\varphi\rangle = \langle\psi\psi|\varphi\varphi\rangle = (\langle\psi|\varphi\rangle)^2.$$

This equation holds only if $\langle\psi|\varphi\rangle = 0$ (orthogonal) or $\langle\psi|\varphi\rangle = 1$ (identical), meaning only for a limited set of states, not for all possible ones.

Implications

- The impossibility of cloning prohibits perfect measurement without disturbing the state.
- It ensures quantum security in cryptography (e.g., the BB84 protocol).
- It highlights the difference between classical and quantum information.

Consider Grover's algorithm

Imagine a phone directory containing N names arranged in completely random order. In order to find someone's phone number with a probability of 50%, any classical algorithm (whether deterministic or probabilistic) will need to look at a minimum of N/2 names. Quantum mechanical systems can be in a superposition of states and simultaneously examine multiple names. By properly adjusting the phases of various operations, successful computations reinforce each other while others interfere randomly. As a result, the desired phone number can be obtained in only $\sqrt{N}$ steps. The algorithm is within a small constant factor of the fastest possible quantum mechanical algorithm.

Grover's algorithm is used for problems in which you need to find a number that satisfies a certain criterion. Moreover, you do not know the number itself, you can only check whether it meets it or not.

To check all possible options in an unordered array, you will need N checks in the worst case, that is, the complexity of the algorithm will be linear O(N).

Grover's quantum algorithm allows you to find the answer in $O(\sqrt{N})$, where N is the number of options. If the array is ordered, you can use binary search, which is faster than O(N), namely O(log N), but here we will consider an unordered array.

The algorithm starts by preparing a quantum system in a uniform superposition of all possible N states. This is done by applying a Hadamard transform to all qubits starting from the zero state, creating a state where each item is equally probable.
Technically: a Hadamard gate is implemented by applying certain electromagnetic pulses to the qubits.
Next, an operation called the oracle is applied. The oracle is a quantum subroutine that recognizes the solution by flipping the sign of the amplitude corresponding to the correct state, while leaving all other states unchanged. This flipping does not measure or reveal the solution, but it marks it subtly in the quantum state's structure.

After the oracle marks the solution, the algorithm applies another operation known as the diffusion operator. The diffusion operator amplifies the amplitude of the marked state and reduces the amplitude of the other states. Essentially, it reflects the quantum state over the average amplitude, increasing the probability of observing the correct answer upon measurement.

The key insight behind Grover's algorithm is that applying the oracle and diffusion operator together acts like a rotation in a two-dimensional space defined by the correct state and the superposition of incorrect states. Each application of this combined operation (called a Grover iteration) brings the system's state closer to the solution. By repeating this process about sqrt(N) times, the probability of measuring the correct answer becomes very high.

At the end of the iterations, a measurement is performed on the quantum system, and with high probability, the result will be the correct solution. Geometrically, Grover's algorithm can be seen as iteratively rotating the system's state vector toward the solution vector, using quantum superposition and interference to efficiently find the desired item among many possibilities.

You repeat the Oracle + Diffusion steps a certain number of times — about N times. Each time, the system's quantum state becomes "more certain" toward the correct answer.

Measure the qubits:
Finally, you measure the qubits.
Physically, this means applying a detector (for example, a laser pulse that reads the qubit's state).
Because of the earlier steps, when you measure, you will most likely get the correct answer.

Algorithm

(i) Initialize the system to the distribution: $\left(\frac{1}{\sqrt{N}}, \frac{1}{\sqrt{N}}, \frac{1}{\sqrt{N}} \ldots \frac{1}{\sqrt{N}}\right)$, i.e. there is the same amplitude to be in each of the N states. This distribution can be obtained in $O(\log N)$ steps, as discussed in section 1.2.

(ii) Repeat the following unitary operations $O(\sqrt{N})$ times (the precise number of repetitions is important as discussed in [BBHT96]):

(a) Let the system be in any state S:

In case $C(S) = 1$, rotate the phase by π radians;

In case $C(S) = 0$, leave the system unaltered.

(b) Apply the diffusion transform D which is defined by the matrix D as follows:

$D_{ij} = \frac{2}{N}$ if $i \neq j$ & $D_{ii} = -1 + \frac{2}{N}$.

This diffusion transform, D, can be implemented as $D = WRW$, where R the rotation matrix & W the Walsh-Hadamard Transform Matrix are defined as follows:

$R_{ij} = 0$ if $i \neq j$;

$R_{ii} = 1$ if $i = 0$; $R_{ii} = -1$ if $i \neq 0$.

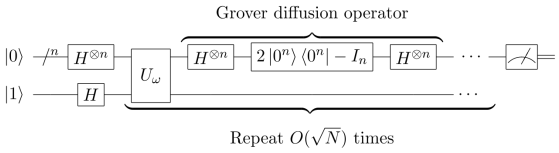
As discussed in section 1.2:

$W_{ij} = 2^{-n/2}(-1)^{\vec{i} \cdot \vec{j}}$, where $\vec{i}$ is the binary representation of i, and $\vec{i} \cdot \vec{j}$ denotes the bitwise dot product of the two n bit strings $\vec{i}$ and $\vec{j}$.

(iii) Sample the resulting state. In case $C(S_v) = 1$ there is a unique state S_v such that the final state is S_v with a probability of at least $\frac{1}{2}$.

Note that step (ii) (a) is a phase rotation transformation of the type discussed in the last paragraph of section 1.2. In a practical implementation this would involve one portion of the quantum system sensing the state and then deciding whether or not to rotate the phase. It would do it in a way so that no trace of the state of the system be left after this operation (so as to ensure that paths leading to the same final state were indistinguishable and could interfere). The implementation does *not* involve a classical measurement.

Quantum circuit of Grover's algorithm



(Bender2k14 • CC BY-SA 3.0)

A quantum circuit is like a recipe showing how quantum bits (qubits) are changed step-by-step using gates.

Reading a quantum circuit is similar to reading sheet music — left to right (usually), showing how each qubit evolves.

Here's a basic guide:

---

1. Each horizontal line = one qubit.
   - Labeled, for example, $|0\rangle$, q_0, q_1, etc.

2. Leftmost state = starting state.
   - Often, all qubits start at $|0\rangle$ (ground state).

3. Gates = boxes or symbols placed on lines.
   - A gate changes the state of the qubit(s) it touches.

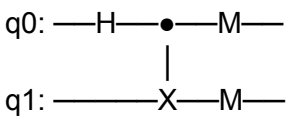
4. You move left to right to follow the flow of operations.

5. Single-qubit gates affect only one qubit.
   - Examples:
     - X = NOT gate (flips $|0\rangle \leftrightarrow |1\rangle$)
     - H = Hadamard gate (makes superposition)
     - Z, Y, etc.

6. Multi-qubit gates involve more than one qubit.
   - CNOT (Controlled-NOT):
     - Control qubit (marked with a dot) decides if the target qubit (marked with ⊕) flips.
   - SWAP:
     - Swaps the states of two qubits.

7. Measurement (often at the end) shows the result.
   - Usually shown with a little meter symbol or an arrow to a classical bit.

Example:



How to read this:
- Start: both qubits at $|0\rangle$.

- Apply Hadamard (H) to q0: it becomes a superposition.
- Then, CNOT gate: q0 controls, q1 targets. (If q0=1, q1 flips.)
- Then measure (M) both q0 and q1.

Sources:

“Quantum Computation and Quantum Information”, Isaac Chuang and Michael Nielsen.

“Quantum Computing for Computer Scientists”, Mirco A. Mannucci and Noson S. Yanofsky.

“Quantum Computing since Democritus”,
Scott Aaronson.

“Quantum Computing for Everyone”, Chris Bernhardt.

“Introduction to Classical and Quantum Computing”, Thomas G. Wong.

“Quantum Supremacy”, Michio Kaku.

IBM Spectrum, September 2024 issue.

“Introduction to Quantum Mechanics”, David J. Griffiths.

“Atomic physics”, Max Born.

“A fast quantum mechanical algorithm for database search”, Lov K. Grover.

Undecidable problems

These theorems and problems demonstrate the limits of the algorithmic approach and confirm that in certain mathematical and scientific tasks, there is no universal algorithm for solving them.

1. Gödel’s Incompleteness Theorem proves that in any sufficiently strong formal system, there are truths that cannot be proven algorithmically. In other words, there is no general algorithm to verify all formulas in first-order arithmetic.

2. Turing's Halting Theorem states that there is no universal algorithm that can determine whether any given program will eventually halt.

3. The Entscheidungsproblem (decision problem) asserts that it is impossible to create a universal algorithm to resolve all questions of first-order arithmetic.

4. Matiyasevich's Theorem on Hilbert's Tenth Problem proves that there is no general method for determining the solvability of arbitrary Diophantine equations.

5. Tarski's Undefinability Theorem shows that the notion of arithmetic truth cannot be expressed within arithmetic itself.

6. The Ruffini-Abel Theorem states that for algebraic equations of degree five or higher, it is impossible to find their roots using only basic mathematical operations such as addition, multiplication, or radicals. There exist equations that cannot be solved by any general algebraic method. This illustrates the limits of what can be solved by algorithms.

7. Kolmogorov Complexity demonstrates the limits of algorithmic description of objects, showing that for many objects, there is no simple algorithm capable of producing the shortest description.

8. Berger's Theorem concerns the problem of finding an infinite regular tiling (tessellation) that can cover a flat surface without gaps or overlaps. Berger proved that there exists an infinite set of tiles that cannot be repeated infinitely without breaking regularity. In other words, there is no universal method for creating infinite regular coverings of a flat surface. This is an example of geometric problems that are not amenable to algorithmic solution.

9. The Four Color Theorem shows that it is impossible to color any map with fewer than four colors without adjacent regions sharing the same color.

10. Brouwer's Fixed-Point Theorem states that it is impossible to map a sphere onto itself without at least one point remaining fixed.

Proof of Brouwer's Theorem for the Interval

Brouwer's fixed-point theorem states that any continuous function mapping a ball (of any dimension) into itself must have at least one fixed point.

Proof in dimension 1.
We can replace any 1-dimensional ball, up to homeomorphism, with the unit interval $I = [0,1]$.
We need to show that if $f: I \to I$ is continuous, then there exists a point x in I such that $f(x) = x$.

Assume the opposite: suppose
$I = \{x \in I \mid f(x) < x\} \cup \{x \in I \mid f(x) > x\}$.

Since f(1) < 1 and f(0) > 0, both of these sets are non-empty.
Due to the continuity of f, it is easy to verify that both of these sets are open.
However, I is a closed set, which contradicts such a partition.
Hence, the assumption is false, and there exists a point x for which f(x) = x.

11. The Three-Body Problem illustrates the complexity of predicting motions in celestial mechanics, representing a case of algorithmic impossibility to solve this class of problems exactly.

Karl Sundman, in 1877, found an exact solution to the three-body motion in the form of infinite series for motion in three-dimensional space. His solution showed that the motion of three bodies can be described using Taylor series, although with certain limitations. However, this analytical solution is only approximate, and a precise description of the three-body motion over long time intervals remains a complex task.

The world of quantum mechanics, according to the Copenhagen interpretation, is not subject to deterministic algorithms, only probabilistic ones. Also we have No-cloning theorem and Heisenberg's uncertainty principle.

In philosophy, Kant’s four antinomies remain unresolved.

Programming languages

There is a programming language called Elixir.
In Elixir, everything is an expression. An expression is evaluated to produce a value, whether it’s a variable, arithmetic operation, or function call. This is different from languages that distinguish between expressions (which return values) and statements (which perform actions).
There aren’t traditional statements in Elixir. Even things like conditionals (e.g., if, switch-case) are expressions and return values.
Elixir does have a garbage collection mechanism, but it’s not implicit in the way that it’s often handled in languages like Java or Python. The Erlang VM (BEAM) handles garbage collection in a more granular way, with each process having its own heap. This means that garbage collection happens per process and doesn’t block the system, leading to lower latency and more predictable performance.
Elixir is dynamically typed, meaning types are not declared explicitly. Instead, types are inferred at runtime, and you can write code without needing to declare types. But you can have some type annotations, if you wish.
Elixir is considered "strongly typed" because the types of variables are strictly enforced at runtime (e.g., adding an integer and a string will raise an error). There is no traditional type inference as in languages like Haskell, but Elixir's dynamic typing still allows flexibility while ensuring type correctness.

Elixir does not support lazy evaluation by default. Evaluation happens eagerly, meaning expressions are computed when they are encountered. However, you can achieve lazy evaluation through explicit constructs, such as using Stream to build lazy sequences.
Elixir emphasizes immutability and side-effect-free functions. Functions in Elixir are considered pure if they always return the same output for the same input and do not modify any state.
Elixir doesn't have traditional loops like 'for' or 'while'. Instead, it uses recursion for repeating tasks. This is a hallmark of functional programming.
Functions in Elixir can call themselves to solve problems that would traditionally require looping. The language encourages recursion, often with tail call optimization to avoid stack overflow errors.
Elixir supports closures. A closure is a function that captures the lexical environment in which it is defined. This allows functions to retain access to variables from their surrounding scope even after the scope has exited.
You can partially apply functions using anonymous functions or using higher-order functions to create curried functions.

In Elixir, processes are lightweight, isolated units of execution that can communicate with each other through message passing. Processes are used for concurrency, and each process can trigger events, such as sending messages or reacting to incoming messages. These events are managed through the message queue and can be handled asynchronously.
Elixir's concurrency model is based on the Actor model, where each process has its own state and does not share state with other processes.
Elixir is designed with the "let it crash" philosophy, where processes are allowed to fail but are isolated so that the rest of the system remains unaffected.

Functions in Elixir are first-class citizens, meaning they can be passed as arguments, returned as values, and assigned to variables. This enables high-order functions, which take functions as parameters or return them.
Elixir supports anonymous functions using the fn keyword. You can create an anonymous function inline and pass it around just like any other value.

In conclusion, Elixir is a powerful functional language built on the Erlang VM with a focus on concurrency, immutability, and high scalability. Its functional nature emphasizes pure functions, recursion, first-class functions, and lightweight processes, making it a good choice for distributed systems. Its syntax and design borrow from various programming paradigms but aim to provide a productive and fault-tolerant programming environment.

Memory leaks in Elixir are relatively rare due to the way the Erlang Virtual Machine (BEAM) handles memory management and garbage collection. However, they can still occur, particularly if you're not careful about managing processes and their state. In Elixir, memory leaks typically arise from improper handling of processes, especially when processes hold onto resources or data that they no longer need.

The syntax of Elixir can be described by a context-free grammar (CFG), which is a formal grammar used to define the syntax of programming languages. Elixir's grammar is context-free, which means that the structure of expressions and statements can be defined by rules that do not depend on the context in which they occur.

Backus-Naur Form is a formal way of describing the syntax of languages. While Elixir's syntax could be described using BNF, the language itself does not directly require you to work with BNF. However, the compiler uses a similar formal grammar to understand and parse Elixir code.

Elixir code is first parsed into an Abstract Syntax Tree (AST). The AST represents the hierarchical structure of the code, and it's this tree that is analyzed by the compiler to generate executable code. You can manipulate the AST directly in Elixir for metaprogramming tasks, such as writing macros.

Elixir provides support for regular expressions through the Regex module. This allows you to perform pattern matching on strings, similar to other languages, enabling text parsing, validation, and other string manipulations.

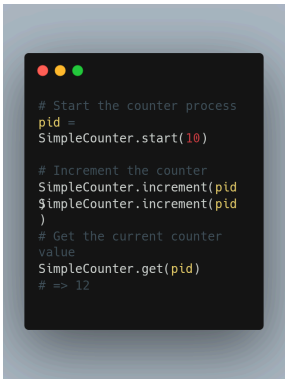
Elixir has supervision trees and is built around the "let it crash" philosophy, meaning a process can fail without affecting others. Go does not have native fault tolerance mechanisms like Elixir, though you can build resilience with goroutines and channels.

Elixir provides true concurrency with lightweight processes that run independently and communicate via message passing. For example, Node.js, while non-blocking, uses a single-threaded event loop and callback-based asynchronous programming (or promises/async-await).

Elixir does not have classes or inheritance. Instead, it uses modules to group related functions and data. A module in Elixir is somewhat analogous to a class in OOP, but it doesn't have any of the typical OOP concepts like inheritance or polymorphism.
Elixir achieves polymorphism-like behavior through pattern matching. Pattern matching allows you to write functions that behave differently based on the shape or content of the data passed to them. This can be used to create behavior that changes depending on the data, much like polymorphism in OOP.
Since Elixir does not use objects, it doesn't have traditional encapsulation mechanisms like private attributes. However, Elixir can encapsulate data within a struct or use modules to create public and private functions.
Elixir emphasizes message passing between processes (lightweight concurrent entities) rather than method calls between objects. Each process in Elixir is isolated, and they can communicate via messages, which makes Elixir well-suited for concurrent, fault-tolerant systems.

You can create a simple counter using just processes and messages.

```elixir
defmodule SimpleCounter do
  def start(initial_value \\ 0) do
    spawn(fn -> loop(initial_value) end)
  end

  defp loop(value) do
    receive do
      {:increment} ->
        loop(value + 1)

      {:get, caller} ->
        send(caller, {:counter_value,
value}) loop(value)
    end
  end

  def increment(pid) do
    send(pid, {:increment})
  end

  def get(pid) do
    send(pid, {:get, self()})

    receive do
      {:counter_value, value} -> value
    end
  end
end
```



```elixir
# Start the counter process
pid =
SimpleCounter.start(10)

# Increment the counter
SimpleCounter.increment(pid
$impleCounter.increment(pid
)
# Get the current counter
value
SimpleCounter.get(pid)
# => 12
```

Explanation:

start/1 creates a new process running loop/1 with the initial counter value.

increment/1 sends a message to increment.

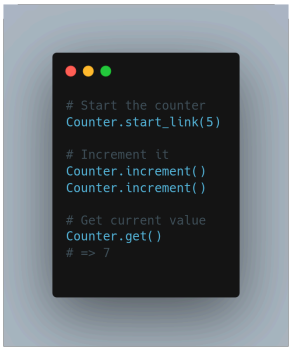
get/1 sends a message to get the value and waits for the reply.

Example with GenServer

GenServer is a core Elixir abstraction that makes it easy to create processes which hold internal state, handle messages, and interact with other parts of the system in a structured way. Instead

of manually spawning processes and writing raw `receive` loops, you implement standardized callback functions like `handle_call` and `handle_cast`, making the code more organized, fault-tolerant, and easier to supervise. GenServer is the foundation for building reliable background services, stateful systems, and concurrent applications in Elixir.
It does this by automatically managing the process lifecycle (starting, receiving messages, updating state, replying), providing standardized callback functions like handle_call and handle_cast, and integrating with the supervision tree to make systems fault-tolerant. Instead of manually spawning processes and writing raw receive loops, you just define how the server should react to different messages. GenServer is the foundation for building reliable background services, stateful systems, and concurrent applications in Elixir.

```elixir
defmodule Counter do
  use GenServer

  ## Client API

  def start_link(initial_value \\ 0) do
    GenServer.start_link(__MODULE__, initial_value, name:
__MODULE__)
  end

  def increment do
    GenServer.cast(__MODULE__, :increment)
  end

  def get do
    GenServer.call(__MODULE__, :get)
  end

  ## Server Callbacks

  @impl true
  def init(initial_value) do
    {:ok, initial_value}
  end

  @impl true
  def handle_cast(:increment, state) do
    {:noreply, state + 1}
  end

  @impl true
  def handle_call(:get, _from, state) do
    {:reply, state, state}
  end
end
```

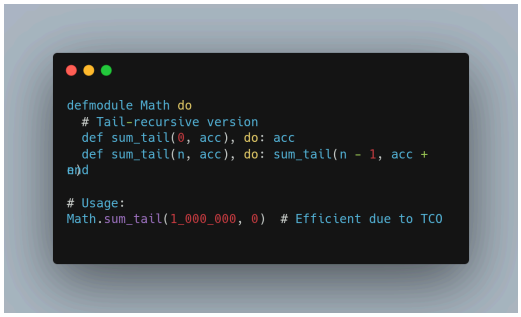


```elixir
# Start the counter
Counter.start_link(5)

# Increment it
Counter.increment()
Counter.increment()

# Get current value
Counter.get()
# => 7
```

Processes in Elixir are **stateful**, **mutable**, and **impure** internally — but they **communicate** purely by **sending messages**.
Each Elixir process is like a little "mutable world," but processes talk to each other in a pure and safe way — just by sending independent messages.
That's why Elixir stays functional, but also practical for real-world systems.

Tail Call Optimization (TCO) in Elixir is a performance feature where tail-recursive functions (those where the recursive call is the last operation) are optimized to reuse the current stack frame, preventing stack overflow and enabling efficient recursion. Elixir supports TCO via the Erlang VM (BEAM).

Here's a brief example of tail call optimization in Elixir:



```elixir
defmodule Math do
  # Tail-recursive version
  def sum_tail(0, acc), do: acc
  def sum_tail(n, acc), do: sum_tail(n - 1, acc +
end

# Usage:
Math.sum_tail(1_000_000, 0)  # Efficient due to TCO
```

The call to sum_tail/2 is the last thing the function does, so the BEAM can optimize it to avoid growing the call stack.

When a tail-recursive function calls itself as the last operation, the Erlang VM (BEAM) doesn't push a new stack frame. Instead, it reuses the current one, effectively turning recursion into iteration.

In Elixir, tail call optimization (TCO) also benefits processes, especially those using recursive loops to maintain state — a common pattern in Elixir's concurrent programming.

This is how GenServers, agents, and other OTP abstractions work under the hood: a looping, tail-recursive function keeps the process alive and responsive without consuming memory over time.

The BEAM VM (Bogdan/Björn’s Erlang Abstract Machine) is the core runtime engine behind Elixir and Erlang, designed for building highly concurrent, fault-tolerant, and distributed systems.

Originally developed for telecom systems, it has become the foundation for modern scalable applications that require reliability and soft real-time behavior.
BEAM is a process-oriented virtual machine optimized for massive concurrency and robustness. Unlike traditional VMs that rely on threads or OS processes, BEAM runs its own lightweight processes that are completely isolated and communicate via asynchronous message passing. This enables the system to spawn millions of processes with low memory and CPU overhead. These processes:
-Don’t share memory.
-Communicate via message passing.
-Are isolated, so one crashing process won’t crash the whole system.
This makes Elixir ideal for building scalable and resilient systems (like chat apps, telecommunication switches, or real-time analytics).
One of BEAM’s most powerful features is its fault-tolerance model, based on the “let it crash” philosophy. Instead of preventing all errors, BEAM encourages developers to isolate errors in small processes, which are automatically supervised and restarted. This is a core idea in the OTP framework, which provides robust abstractions like supervisors, gen_servers, and applications to structure systems safely.
Another standout capability is hot code swapping — the ability to update code in a live system without shutting it down. BEAM can hold two versions of a module simultaneously, ensuring seamless upgrades with zero downtime. This is critical for systems that require 24/7 uptime, such as telecom switches, banking systems, or distributed web services.
BEAM can keep two versions of a module in memory at the same time: the current and the old version.
When you load a new version of a module:
-Existing processes keep running the old version.
-New processes or function calls will use the new version.
-Once all processes stop using the old version, it is garbage collected.

BEAM also natively supports distribution: multiple BEAM nodes (on the same or different machines) can connect and communicate as if they were part of the same system. This makes Elixir ideal for clustered deployments and real-time communication between services.

Under the hood, BEAM optimizes functional programming features such as immutability, tail-call recursion, and pattern matching. Since Elixir compiles down to BEAM bytecode, it inherits all these runtime benefits. The VM is also garbage-collected and uses per-process heaps, ensuring that memory issues in one process do not affect others.

The BEAM VM was primarily developed by Bogumil "Bogdan" Hausman and Björn Gustavsson at Ericsson in the 1990s as part of the Erlang/OTP system for building reliable telecom software. The predecessor of the BEAM was JAM (Joe's Abstract Machine), which was the first virtual machine for the Erlang language and was written by Joe Armstrong and Mike Williams in the C language.

Here is a list of programming languages that introduced brand-new ideas to programming, changing how developers think or work:

0. Lambda calculus (1940)
New Idea: Anonymous functions, Closure, Currying.

1. Lisp (1958)

New Idea: Code as data (homoiconicity), recursion as a core concept, garbage collection, closure.

2. ALGOL (1958–1960)

New Idea: Structured programming, block structure with scoped variables, Backus-Naur Form (BNF).

3. Simula (1967)

New Idea: First object-oriented programming (classes, inheritance).

4. Smalltalk (1972)

New Idea: Pure object-oriented programming, everything is an object.

5. Prolog (1972)

New Idea: Logic programming, declarative paradigm based on formal logic.

6. ML (1973)

New Idea: Hindley-Milner type inference, algebraic data types, pattern matching.

7. Eiffel (1985)
New Idea: Design by Contract.

8. Haskell (1990)

New Idea: Pure functional programming, lazy evaluation by default, monads for handling effects.

9. Erlang (1986)

New Idea: Lightweight process concurrency, fault-tolerant distributed systems.

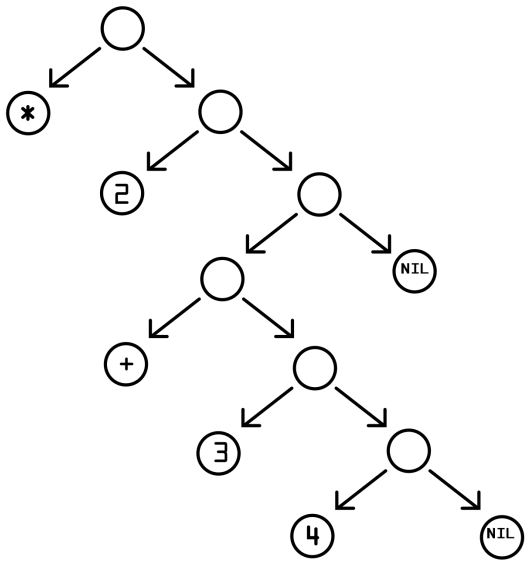
10. Self (1987)

New Idea: Prototype-based object-oriented programming.

11. SPARK (1990s, based on Ada)
New Idea: Formal verification integrated into a general-purpose language.

12. Rust (2010)
New Idea: Ownership and borrowing system for memory safety without garbage collection; fearless concurrency, zero-cost abstractions.

S-expression (or symbolic expression, abbreviated as sexpr or sexp) is an expression in a like-named notation for nested list (tree-structured) data. S-expressions were invented for, and popularized by, the programming language Lisp, which uses them for source code as well as data.



Tree data structure representing the S-expression (* 2 (+ 3 4))

Sources:

1. Jean E. Sammet, Programming languages: History and Fundamentals, 1969
2. Niklaus Wirth, Algorithms + Data Structures = Programs, 1976

3. John Hopcroft, Jeffrey Ullman, Introduction to Automata Theory, Languages, and Computation, 1979 (first edition)
4. Harold Abelson, Gerald Jay Sussman, Structure and Interpretation of Computer Programs, 1984
5. Alfred Aho, Jeffrey Ullman, Compilers: Principles, Techniques, and Tools, 1985
6. Brad Cox, Object-Oriented Programming: An Evolutionary Approach, 1986
7. Bertrand Meyer, Object-Oriented Software Construction, 1988
8. Thomas J. Bergin, History of Programming Languages, circa 1996–1997
9. John Hopcroft, Jeffrey Ullman, Introduction to Automata Theory, Languages, and Computation, 2000 (second edition)
10. Joe Armstrong, Programming Erlang: Software for a Concurrent World, 2007
11. Bjarne Stroustrup, Programming: Principles and Practice Using C++, 2009
12. Joe Armstrong, Programming Erlang (2nd edition), 2013
13. Bruce Tate, Seven Languages in Seven Weeks, 2014
14. Douglas Crockford, How JavaScript Works, 2018
15. Dan Vanderkam, Effective TypeScript: 62 Specific Ways to Improve Your TypeScript, 2019
16. Axel Rauschmayer, Tackling TypeScript: Upgrading from JavaScript, 2021
17. Axel Rauschmayer, JavaScript for Impatient Programmers, 2022

Information Theory and Signal Transmission

A radiowave is a stream of photons. It is also an electromagnetic wave, described by Maxwell's equations.
A radiowave can be influenced by gravity and magnetic fields.
Radiowaves can interfere, reflect, and diffract. They obey Snell's law.

A radiowave is a transverse wave that can have a polarization.

The phase of a wave is a physical characteristic that defines the state of oscillation at a given moment in time and at a specific point in space. It indicates how much a point on the wave is "behind" or "ahead" of another point in its oscillations.
The phase is usually denoted by the Greek letter φ (phi) and is expressed in radians or degrees. It can be represented as part of the argument of trigonometric functions that describe the wave, for example:
$y(x, t) = A * \sin(kx - \omega t + \varphi_0)$, where:
- A — amplitude of the wave;
- kx — spatial component (wave number $k = 2\pi / \lambda$, where λ is the wavelength);
- -ωt — temporal component ($\omega = 2\pi f$, where f is the frequency of the wave);
- φ_0 — initial phase.

The phase is important in wave interference phenomena, where the interaction of waves depends on the phase difference between them. If the phases of the waves coincide, constructive interference occurs, and if they are opposite, destructive interference occurs.

Fourier Transform and Modulation

The Fourier transform is a generalization of the Fourier series, which in its most general form introduces the use of complex functions. An example of applying the Fourier transform is determining the constituent tones in a sound wave. The Fourier transform allows one to transition from representing a function f(x) in the "time-amplitude" domain to its representation in the "frequency-amplitude" domain. The transform is named after the French mathematician Jean-Baptiste Joseph Fourier, who introduced the concept in 1822.

Fourier Transform:

$$\hat{f}(\xi) = \int_{-\infty}^{\infty} f(x)\, e^{-i2\pi\xi x}\, dx.$$

In the function f'(ξ), frequency is used as the argument, and the result is a complex number whose real part indicates the amplitude of the frequency, while the imaginary part indicates its phase. If the amplitude for a certain frequency is zero, this means that the original function does not contain a component of that frequency. If the amplitude for a given frequency is positive, this means that the original function f(x) contains a component of that frequency, and the phase shows the phase of this component. If the amplitude is negative, it also indicates that f(x) contains a component of that frequency, but with the opposite phase.

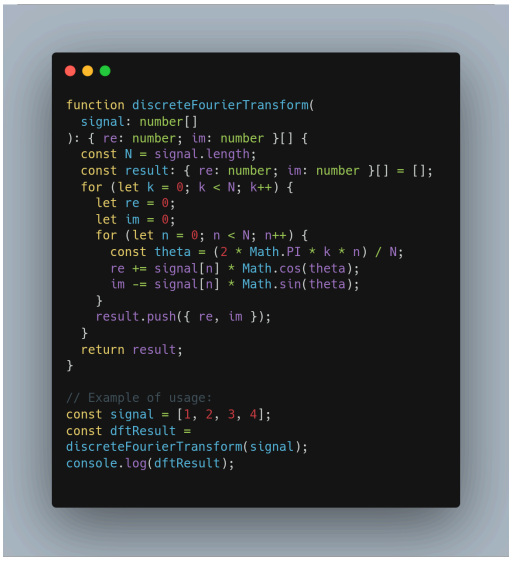
Thus, by analyzing the amplitudes and phases of the components in the Fourier transform, one can determine which frequencies are present in the original function and which are not. This information can be used to analyze the structure of the function f(x) and its spectrum.

Formula components:

$e^{-i2\pi\xi x} = \cos(2\pi\xi x) - i * \sin(2\pi\xi x)$ – Euler's formula.
ξ — frequency in hertz;
f'(ξ) — the Fourier transform, a function that takes a frequency (a real number) and returns a complex number;

f(x) — a real-valued function whose frequency content is being analyzed and which returns the amplitude of oscillations with respect to the time axis x;
2π — the circumference of a circle with diameter 1, measured in radians;
e — Euler's number.

Algorithm of Discrete Fourier Transform in TypeScript



```typescript
function discreteFourierTransform(
  signal: number[]
): { re: number; im: number }[] {
  const N = signal.length;
  const result: { re: number; im: number }[] = [];
  for (let k = 0; k < N; k++) {
    let re = 0;
    let im = 0;
    for (let n = 0; n < N; n++) {
      const theta = (2 * Math.PI * k * n) / N;
      re += signal[n] * Math.cos(theta);
      im -= signal[n] * Math.sin(theta);
    }
    result.push({ re, im });
  }
  return result;
}

// Example of usage:
const signal = [1, 2, 3, 4];
const dftResult =
discreteFourierTransform(signal);
console.log(dftResult);
```

The `discreteFourierTransform` function computes the Discrete Fourier Transform (DFT) of an input signal and returns an array of objects, where each object has two properties: `re` (real) and `im` (imaginary).

Here's what the function output represents:
- re (real): the real part of a complex number corresponding to a specific frequency component in the DFT. It represents the amplitude or strength of that frequency component in the output signal.
- im (imaginary): the imaginary part of a complex number corresponding to a specific frequency component in the DFT. It represents the phase information of that frequency component in the output signal.

The DFT decomposes the signal into a sum of sinusoidal components (sine and cosine) at various frequencies. The values of `re` and `im` for each frequency component provide information about how much of that specific frequency exists in the output signal and its phase relative to the original signal.

Look for the `re` and `im` values with the highest amplitude, as these indicate the presence of strong frequency components in the input signal.

A complex number can be represented as an ordered pair of two real numbers:

`<a, b>`, where a complex number with a zero second component is considered a real number, e.g., `<2, 0>` = 2.
Two complex numbers are equal if their components are equal in value and order.

Adding complex numbers:
<2,3> + <4,5> = <2+4, 3+5>.

Multiplication of complex numbers:
<2,3> * <4,5> = <2*4-3*5, 2*5+3*4>.

The imaginary unit can be specified as a complex number:

i = <0, 1>;
i^2 = -1;
Proof:
<0, 1> * <0, 1> = <0*0-1*1, 0*1+1*0> = <-1, 0> = -1;

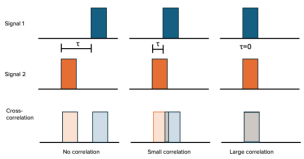
The FFT `y[k]` of length N of the length-N sequence `x[n]` is defined as

$$y[k] = \sum_{n=0}^{N-1} e^{-2\pi j \frac{kn}{N}} x[n] \,,$$

and the inverse transform is defined as follows

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} e^{2\pi j \frac{kn}{N}} y[k] \,.$$

Cross-correlation is a method used to measure how similar two signals are at different time shifts. It helps to find patterns or the time delay between two signals.



Cross-correlation compares two signals by shifting one signal over the other and checking how much they match at each time shift. It tells you if one signal is a delayed version of another or how closely they resemble each other at different points in time.

Formula: In continuous time, cross-correlation between two signals x(t) and y(t) at a time shift tau is calculated as: Rxy(tau) = integral from -infinity to +infinity of x(t) * y(t+tau) dt In discrete time, cross-correlation is calculated as: Rxy[n] = sum over all k of x[k] * y[k+n] Here, tau or n represents the time shift, and x(t) and y(t) are the two signals being compared.
When the cross-correlation value is high, it means the two signals are most similar at that specific time shift.
If the cross-correlation value is low, the two signals don't match well at that time shift.

Where it's used:
Signal alignment: If one signal is delayed or distorted, cross-correlation can be used to align the signals.
Pattern matching: In applications like image processing or time series analysis, cross-correlation helps to identify patterns or trends by comparing signals at various time shifts.
Audio processing: Cross-correlation is used to detect and match sounds, for example, in music or speech recognition systems.
Radar/Seismic detection: It's used to detect echoes or delayed signals in radar systems or seismic data analysis.

In simple terms, cross-correlation helps you find how much one signal matches another, and at which time shift they match best.

In continuous time, cross-correlation between signals $x(t)$ and $y(t)$ at a time shift τ is:

$$R_{xy}(\tau) = \int_{-\infty}^{\infty} x(t)y(t+\tau)dt$$

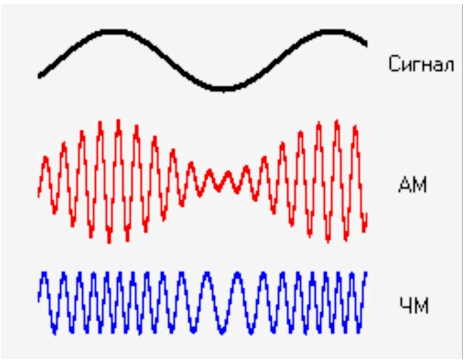
In discrete time, it's:

$$R_{xy}[n] = \sum_{k} x[k]y[k+n]$$

3. **What it tells you**:
   - When the cross-correlation value is high, it means the two signals are similar at that time shift.
   - If the value is low, the signals don't match at that shift.

Modulation is the process of altering the parameters of a high-frequency radio signal (the carrier) according to the characteristics of a low-frequency signal that carries the information.

This is necessary for transmitting information over long distances via radio waves, as the low-frequency signal does not have enough energy to travel far on its own.



Transmission of a Sinusoidal Signal via Amplitude Modulation (AM) and Frequency Modulation (FM)

The main parameters of the carrier signal that can be altered are amplitude, frequency, and phase. Accordingly, there are three main types of modulation:

1. Amplitude Modulation (AM)
   In this type of modulation, the amplitude of the carrier changes in proportion to the amplitude of the information signal. The frequency and phase remain unchanged. AM is used in broadcasting, for example, for transmitting signals on medium-wave (MW) radio.

2. Frequency Modulation (FM)
   In FM, the frequency of the carrier changes depending on the amplitude of the information signal. This type of modulation provides better sound quality and greater resistance to interference, which is why it is widely used in FM radio.

3. Phase Modulation (PM)
   In PM, the phase of the carrier changes according to the information signal. This method is often used in digital communication systems.

Modulation allows efficient transmission of information by increasing bandwidth, reducing signal interference, and optimizing use of the radio frequency spectrum.

Example of Amplitude Modulation (AM):

Imagine you want to transmit your voice over the radio. Your voice is a low-frequency signal (e.g., 300 Hz to 3 kHz), which cannot travel long distances on its own. To enable transmission, a high-frequency carrier signal (e.g., 1 MHz) is used.

During AM:
- The amplitude of the carrier signal (a sinusoidal wave at 1 MHz) is modified according to the amplitude of your voice.
- The carrier frequency remains constant (1 MHz), but its "strength" (amplitude) increases or decreases depending on how loud your voice is.

Graphically:
- The carrier appears as a constant high-frequency sine wave.
- The modulated signal appears as a sine wave whose amplitude outlines the shape of your voice signal.

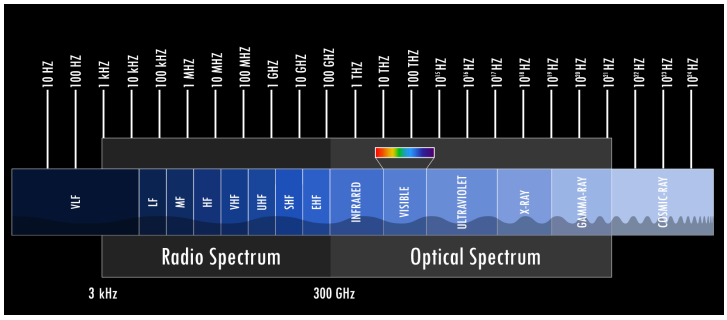
Real-life example:
AM is used in medium-wave (MW) radio. If you tune in to 1000 kHz (1 MHz), you hear a broadcast that was modulated in this way.

Example of Frequency Modulation (FM):

Suppose you want to transmit music over FM radio. Your signal is again a low-frequency signal (e.g., a musical waveform). In FM:
- The frequency of the carrier (e.g., 100 MHz) changes depending on the amplitude of the music signal.
- The carrier amplitude remains constant.

Real-life example:
FM radio at 100 MHz transmits sound with better quality, as FM is more resistant to interference.

Ultra high frequency (UHF) is the ITU designation for radio frequencies in the range between 300 megahertz (MHz) and 3 gigahertz (GHz), also known as the decimetre band as the wavelengths range from one meter to one tenth of a meter (one decimetre).
Super high frequency (SHF) is the ITU designation for radio frequencies (RF) in the range between 3 and 30 gigahertz (GHz).
Frequencies in the SHF range are often referred to by their IEEE radar band designations: S, C, X, Ku, K, or Ka band, or by similar NATO or EU designations.



(Image from nasa.gov).

Institute of Electrical and Electronics Engineers (IEEE) Radar Band Designations

3-30 MHz,
Wavelength: 100-10 m,
Band: HF.

30-300 MHz,
Wavelength: 10–1 m,
Band: VHF.

300-1000 MHz,
Wavelength: 100–30 cm,
Band: UHF.

1-2 GHz,
Wavelength: 30–15 cm,
Band: L.

2-4 GHz,
Wavelength: 30–15 cm,
Band: S.

4-8 GHz,
Wavelength: 15–7.5 cm,
Band: C.

8-12 GHz,
Wavelength: 7.5–3.75 cm,
Band: X.

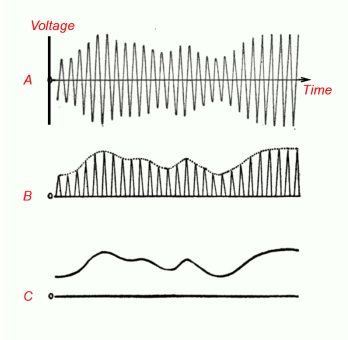
Crystal Detector:

A crystal detector was an early device used to extract an audio signal from a radio signal, working due to the unidirectional conductivity of semiconductors. Its construction included a crystal (e.g., galena) and a fine metal wire (a "cat's whisker"). The detector allowed current to flow in only one direction, enabling separation of the low-frequency audio signal from the high-frequency carrier wave.

Modern diode:

A diode performs the same function using a semiconductor p-n junction. It efficiently conducts current in only one direction and has replaced the crystal detector due to its greater reliability, stability, and compactness. Diodes don't require the delicate contact adjustment that "cat's whiskers" did, and are standard components in modern electronics.

When detecting an AM signal, the diode (or crystal detector) cuts off one half of the signal, producing a pulsating direct current. A bypass capacitor then smooths the pulsations, leaving the audio signal used to generate sound. This makes the diode a more practical choice in modern devices.



(Amplitude modulation detection, Public domain)

A diagram showing how a crystal detector works. In diagram A, an amplitude-modulated radio signal from the receiver’s tuned circuit is shown, which is applied as voltage to the contacts of the detector. The rapid oscillations correspond to the radio frequency carrier wave. The audio signal (sound) is contained in the slow changes (modulation) of the wave’s amplitude. If this signal were applied directly to the headphones, it wouldn’t be able to turn into sound, as the amplitude shifts are symmetric relative to the axis, averaging zero, which would result in no movement of the headphone diaphragm.

In diagram B, the current passing through the crystal detector and being supplied to the headphones and bypass capacitor is shown. The crystal conducts current only in one direction, clipping the oscillations on one side of the signal, leaving a pulsating DC, the amplitude of which does not equal zero and changes in accordance with the audio signal.

In diagram C, the current passing through the headphones is shown. The bypass capacitor, connected in parallel to the headphone terminals, along with the diode’s internal forward resistance, creates a low-pass filter that smooths the waveform, eliminating the radio-frequency carrier’s pulses and leaving the audio signal. When this alternating current passes through the piezoelectric crystal of the headphone, it causes deformation (bending) of the crystal, which deflects the headphone diaphragm; these varying deflections cause the diaphragm to vibrate and create sound waves (acoustic waves). If instead, a speaker with a voice coil is used, the

alternating current from the low-pass filter passes through the voice coil, generating a changing magnetic field that attracts and repels the diaphragm, again causing it to vibrate and produce sound.

FMCW radar (Frequency-Modulated Continuous-Wave) transmits a continuous signal whose frequency changes over time (e.g., increases linearly). When the signal reflects off an object and returns, the difference between the transmitted and received signal frequencies allows the distance to the object (via delay) and its speed (via Doppler frequency shift) to be determined. This type of radar is well-suited for short and medium ranges, with high accuracy in distance measurement, and is commonly used in vehicles, drones, and industrial sensors. It is also compact, energy-efficient, and capable of tracking multiple objects simultaneously due to signal processing in the form of a range-speed map.

Pulse-Doppler radar operates by sending short, powerful pulses and measuring the time it takes for the reflected signal to return—this allows the distance to be determined. Additionally, analyzing the phase shift between the pulses provides information about the object's speed. This radar works excellently at long ranges and can accurately detect fast-moving objects, distinguishing them from stationary ones (such as background or ground). It is used in military applications, aviation, air defense, and meteorology, where tracking a large number of objects over a wide area is required.

In FMCW radar, the Doppler effect manifests as an additional frequency shift in the reflected signal when an object is moving. Since the radar continuously transmits a signal with a varying frequency (e.g., linearly increasing—"chirp"), when reflected from the object, the radar receives a copy of this signal with a time delay (corresponding to distance) and, if the object is moving, also with a shifted frequency (due to the Doppler effect).

To separate these two effects—the delay (distance) and Doppler shift (speed), FMCW radar typically transmits several consecutive "chirps" and analyzes the phase change between them. This allows the Doppler frequency (shift due to motion) to be calculated separately from the frequency that arises due to the delay. Thus, a moving object causes a continuous phase shift between the "chirps", enabling accurate speed measurement.

The sampling theorem is a fundamental principle in digital signal processing that pertains to the discretization of analog signals. It formulates the conditions under which an analog signal can be accurately reconstructed from its discrete samples. In the context of radar and modulation, the theorem plays an important role as these fields deal with signals transmitting information via electromagnetic waves.

According to the theorem, if an analog signal is band-limited (i.e., it has a limited frequency range), it can be uniquely reconstructed from its discrete samples, provided that the sampling frequency (fs) exceeds twice the maximum frequency of the signal (f_{max}). This criterion is known as the Nyquist-Shannon criterion. If the sampling frequency is below this threshold, aliasing occurs, which distorts the signal and makes reconstruction impossible.

Shannon–Hartley Theorem on Channel Capacity

The Shannon–Hartley theorem was proposed by engineers Claude Shannon and Ralph Hartley in 1948.
The theorem describes how fast information can be transmitted through a channel affected by noise. The core idea is that there exists a maximum transmission rate through a channel with a given noise level that can be achieved without errors.

Key concepts used in the theorem:

- Channel Capacity (C): The maximum rate at which information can be transmitted through a channel without errors, measured in bits per second (bps).
- Noise (N): Noise in a channel leads to loss or distortion of part of the information during transmission.
- Channel Bandwidth (B): The range of frequencies a channel can transmit without information loss.

The Shannon–Hartley theorem is formulated as:

$C = B \times \log_2(1 + S/N)$

Where:
- C – channel capacity in bits per second
- B – bandwidth in hertz
- S – signal power in watts
- N – noise power in watts

A simple example for calculating bandwidth (B) in analog radio or telecommunication systems:

$B = f_{max} - f_{min}$,
where f_{max} is the highest frequency the channel can transmit, and f_{min} is the lowest.
For example, if a channel can transmit signals from 1000 Hz to 5000 Hz, the bandwidth is:
5000 Hz − 1000 Hz = 4000 Hz.

Suppose we have a communication channel with a bandwidth of B = 10 kHz, signal power S = 100 mW, and noise power N = 1 mW.
Then, the channel capacity is:
$C = 10 \text{ kHz} \times \log_2(1 + 100 \text{ mW} / 1 \text{ mW}) = 66.7 \text{ kbps}$.

The higher the B, the more information can be transmitted over time. A higher S/N ratio means the signal is stronger compared to the noise, allowing more reliable extraction of information at the receiver. When the signal-to-noise ratio is low, the noise can significantly distort the signal,

especially at higher frequencies. In such cases, engineers may limit the bandwidth used to reduce noise impact and ensure accurate signal reception.

Capacity of a discrete (digital) noise-free channel:

$C = \log_2(m) \times V_t$,
where m is the base of the signal code (number of symbols in the alphabet), and
V_t is the number of symbols transmitted per unit time.

- For m = 2 (binary code) and V_t = 1000 symbols/sec, the capacity is:
  $C = \log_2(2) \times 1000 = 1 \times 1000 = 1000$ bps

- For m = 4 (a system with four signal levels) and V_t = 1000 symbols/sec, the capacity is:
  $C = \log_2(4) \times 1000 = 2 \times 1000 = 2000$ bps

SNR (Signal-to-Noise Ratio) is the ratio of signal power to noise power in a communication channel. The higher the SNR, the more clearly the useful signal can be "heard" among the noise.

SNR determines how much data can be reliably transmitted through the channel. According to the Shannon–Hartley theorem, the maximum data transmission rate depends on the channel's bandwidth and SNR.

A symbol in a transmission channel is a unit of signal that carries information. For example, in a system with two signal levels (e.g., 0 or 1), each symbol carries 1 bit. If the system can distinguish more levels (e.g., 4 levels), then one symbol can carry more bits—for example, 2 bits, since $\log_2(4) = 2$. The cleaner the signal (the higher the SNR), the more levels can be distinguished, and the more bits each symbol carries.

Bandwidth is the frequency range used by a signal. It determines how many symbols can be transmitted per second. For example, if the bandwidth is 1 Hz, only one symbol can be transmitted per second; if it's 1 kHz—up to a thousand symbols per second. Bandwidth is the difference between the upper and lower frequencies. If your channel transmits only one frequency at one hertz, then the bandwidth is 1–0, that is, 1.

The Shannon formula describes the maximum error-free data transmission rate through a communication channel:

$C = B \cdot \log_2(1 + SNR)$
where:
- C is the maximum data transmission rate (in bits per second),
- B is the bandwidth, or how many symbols can be transmitted per second,

- $\log_2(1 + \text{SNR})$ is the number of bits that can be transmitted per symbol depending on the signal-to-noise ratio (SNR).

The greater the bandwidth and the higher the SNR, the more symbols can be transmitted per second and the more bits each symbol carries—this increases the maximum data transmission rate.

Radio signals are measured in units like dBm and dB because they deal with very large or very small values of power that are hard to express with regular units like watts. For example, a mobile phone might receive a signal as small as 0.00000000001 watts, which is inconvenient to write and understand. Instead, we use a logarithmic scale, which compresses wide ranges of values into manageable numbers. On this scale, 1 milliwatt is defined as 0 dBm. A weaker signal, like 0.00000000001 watts, becomes -80 dBm. This is much easier to read and compare. The decibel (dB) itself is a unit for comparing signal levels. A difference of 10 dB means one signal is 10 times stronger or weaker than another. This makes it easy to understand gain or loss in a system. For example, if your antenna boosts the signal by 20 dB, it means the signal is 100 times stronger. In everyday use, dBm helps us describe absolute signal strength (like the strength of a Wi-Fi signal), while dB helps describe changes or differences in strength (like how much a signal is amplified). This system of measurement matches how technology processes signals and makes analysis and troubleshooting much more practical.

Sampling Theorem

The Sampling Theorem, also known as the Nyquist–Shannon Theorem, is a fundamental principle in signal theory and telecommunications. This theorem establishes the relationship between the sampling rate of a signal, its bandwidth, and the maximum frequency of the signal that can be correctly reconstructed from a digital signal.

The main points of the theorem are approximately as follows:

- Sampling rate (sampling frequency): A signal must be sampled (measured) at least twice as frequently as its maximum frequency component.
  $f_s \geq 2 \times f_{max}$,
  where f_s is the sampling frequency (sampling rate), and f_{max} is the maximum frequency of the signal.

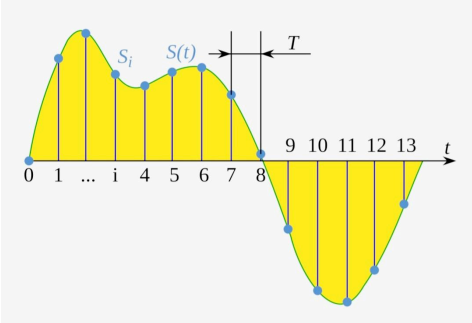
- Bandwidth (B) of the signal: The signal's bandwidth should be less than half the sampling rate.
  $B \leq ½ \times f_s$

These conditions are essential to avoid aliasing, which can lead to information loss and signal distortion during sampling.

Aliasing is a phenomenon that occurs when an analog signal is sampled at an insufficient rate, violating the Nyquist–Shannon theorem. When the signal contains high-frequency components, but the sampling rate is too low, “phantom” frequencies may incorrectly appear during signal analysis.

As a result of aliasing, low-frequency components may be mistakenly interpreted as high-frequency ones, leading to distortion. To avoid aliasing, the sampling rate must be high enough to accurately represent all high-frequency components of the signal. The Nyquist–Shannon theorem defines the minimum required sampling rate to prevent aliasing.

An illustration of signal sampling: a continuous function is shown in green, and the discrete sequence—in light blue.



T – Sampling Interval
The relationship between the sampling interval (T) and the sampling frequency (Fs) is defined as:
T = 1 / Fs
For example, if your sampling frequency is 10 kHz (10,000 samples per second), the sampling interval will be:
T = 1 / 10,000

Information Entropy

Entropy defines the uncertainty or randomness in information. The higher the entropy, the greater the uncertainty.

Shannon's formula for entropy (H) is as follows:
$H(X) = -\Sigma(i=1 \text{ to } n)\ P(x_i) * \log(P(x_i))$

Given a discrete random variable X, which takes values in the alphabet $\{x_n\}$.

In this formula, the logarithm "reduces the weight" of high probabilities, making their contribution smaller, thus emphasizing the importance of less likely events. Probability is defined over the interval [0,1].

In information theory, the entropy of a random variable quantifies the average level of uncertainty or information associated with the variable's potential states or possible outcomes.

Let's say we have a random variable X that can take on three possible values with the following probabilities:
$P(X=1) = 0.4$
$P(X=2) = 0.3$
$P(X=3) = 0.3$

$H(X) = -\Sigma(i=1 \text{ to } 3)\ P(x_i) * \log(P(x_i)) = -(0.4 * (-1.322) + 0.3 * (-1.737) + 0.3 * (-1.737)) \approx 1.57$

If a random variable has three possible outcomes, the maximum entropy will be:
$H_{max} = \log(3) = 1.585$

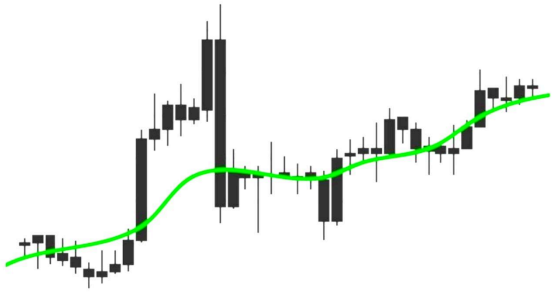
The smaller the probability of an event, the greater its contribution to the surprise (uncertainty), due to the logarithm.

The choice of logarithm base varies depending on the application.
- Base 2 yields the unit "bit" (or "shannon")
- Base e gives "natural units" (nats)
- Base 10 gives units called "dits" or "hartleys"

Hartley (dit) — a logarithmic unit for measuring information or entropy based on base-10 logarithms and powers of 10, rather than powers of 2 and base-2 logarithms that define a bit (or shannon). One hartley corresponds to the amount of information of an event with a probability of 1/10.

Interestingly, Kolmogorov complexity and entropy are related in such a way that in the case of maximum entropy (maximum uncertainty), Kolmogorov complexity is also maximal. In other words, if the information is very random and lacks internal order, it is difficult to compress or describe succinctly.
Thus, entropy and Kolmogorov complexity can serve as interchangeable measures in various contexts within information theory.

A moving average is a filter widely used in signal processing and time series analysis.



MOVING AVERAGE

What does it do?
A moving average smooths out the fluctuations of a signal by calculating the average of several neighboring points. This reduces the impact of noise (rapid fluctuations) and makes the overall trends more visible.

How does it work (simple example):
Sequence: [3, 5, 8, 6, 4, 9] 3-point moving average: [(3+5+8)/3, (5+8+6)/3, (8+6+4)/3, (6+4+9)/3] → `[5.33, 6.33, 6.0, 6.33]`

"The purpose of music lies in pleasing the ear," wrote Leonhard Euler in his book “A New Theory of Music” (1739).
It is well known that Pythagoras and his followers began studying music from a mathematical perspective.
They started to rely not only on hearing but also on mathematical ratios to identify beautiful music. It is said (notably by Nicomachus of Gerasa) that Pythagoras discovered that a string half the length of another sounds harmonious with it. If string A is half the length of string B, it will sound harmonious with B, but its pitch will be an octave higher—meaning it will have twice the frequency of vibration. Pythagoras also reportedly found that a string 2/3 the length of another would still sound consonant with the original.

That is, if you take two identical strings and cut one to 2/3 its original length, the two will still sound harmonious. The interval between the note of the first string and the second is called a fifth (quint), if the ratio of their lengths is 2:3. By building a sequence of fifths, one obtains a 7-step diatonic scale. The discovery of these mathematical relationships between strings showed that there are certain laws governing auditory perception. Later, this led to the concept

of the brain as a device that processes specific vibrations and perceives some of them as pleasant—consonances—and others as unpleasant—dissonances, or even noise.

Based on the Pythagorean tuning system, the equal temperament system was developed in the 16th century. It was more precise than Pythagoras's system because it divided the octave into 12 equal parts rather than 7. For example, the note C of the second octave is twelve semitones away from the note C of the first octave.

Humans can hear sounds with frequencies from 16 Hz to 20 kHz.
We know that the note A of the first octave has a frequency of 440 Hz, and the A of the second octave has a frequency of 880 Hz. The ratio of the string lengths in this case is 1:2. That is, the same string that produces 440 Hz will produce 880 Hz if it is made twice as short. A frequency of 440 Hz means 440 vibrations per second.

To divide the distance between the A of the first and second octaves into 12 equal parts, we need to find a number that, raised to the 12th power, equals 2. That is, if any base frequency is multiplied by this number 12 times, it will be the same as multiplying the frequency by 2.
In the twelve-tone equal temperament system, which divides the octave into 12 equal parts, the size of one semitone (i.e., the ratio of frequencies between two adjacent notes) is the 12th root of 2:
root(2, 12) = 2^(1/12) ≈ 1.059...

The frequency of a note in equal temperament can be calculated using the formula:
$f(i) = f_0 \times 2^{(I/12)}$, where i is the number of semitones between the target note and the base note of frequency f_0.
For example, if we take $f_0 = 440$ Hz (i.e., the note A of the first octave—the 5th fret of the thin E string on a six-string guitar), then the note C of the second octave, which is three semitones above A, is calculated as:
440 × 2^(3/12) = 523.25...

The "Treatise on Musical Flutes" by Sima Qian, a Chinese historian of the early Han dynasty, describes a 12-tone musical scale. This scale roughly corresponds to the modern chromatic scale. It can be constructed as follows:

Let the string length of the first (primary) sound be 1.
Assume the string for the second sound is 0.95 of the length of the first. Then the length of the third sound's string would be 1 × 0.95 × 0.95, and for the fourth: 1×0.95^3, and so on.

Sima Qian builds his tuning system starting with a pipe of length 2/3 of the base pipe, and then multiplies it and its resulting fractions by 4/3 or 2/3.
(This gives the following lengths: 1, 2/3, 8/9, 16/27, 64/81, 128/243, and so on.)

Problem:

You have 7 markers, each in one of the colors of the rainbow. You may use any number of markers — from one to all seven — but each color can be used no more than once. The order in which the colors are used does not matter.

How many different ways can you color an object under these conditions?

(Example: coloring with green and red is one option; coloring with red, blue, and green is another. However, green + red is considered the same as red + green.)

Answer: 127 ways.

$2^7 - 1 = 127$.

Proof:

Why, if |A| = n, then |Pow(A)| = pow(2, n) = 2^n?
As we know, Pow(A) is the set of all subsets of A.
Obviously, the maximum subset in Pow(A) will have a cardinality of |A|, and others will have fewer elements.
Because if an element in Pow(A) had a cardinality greater than that of A, it would mean we're getting new elements from nowhere.
Thus, we can represent every subset from Pow(A) as a binary number of length |A|, placing a 1 in each position where the corresponding element exists.

For example, A = {1, 2, 3}, then
Pow(A) = {{}, {1}, {2}, {3}, {1,2}, {1,3}, {2,3}, {1,2,3}}.

000 - {}
111 - {1,2,3}
100 - {1}
010 - {2}
001 - {3}
110 - {1,2}
101 - {1,3}
011 - {2,3}

Thus, by the formula from combinatorics, we get |Pow(A)| = pow(2, n), where 2 = |{0,1}| and n = |A|.

System engineering and design

Friends, don't confuse System Engineering with System Design, or Software Engineering with Software Design.

These distinctions are clearly described in SWEBOKv3 and SWEBOKv4.

Here's a brief explanation of each term:

1. Software Engineering:
   This is the overall process of creating software, which includes analysis, design, development, testing, maintenance, and management of software systems. Software engineering covers not only technical aspects of development but also managerial, financial, and organizational processes to ensure successful software delivery.

2. Software Design:
   This is a subprocess within software engineering focused on developing the architecture and components of software. At this stage, algorithms, data structures, interfaces, and other details are defined, forming the foundation for further coding. The goal is to create an efficient and scalable program that meets user requirements.

3. System Engineering:
   This is a broader approach to designing and developing complex systems that include both hardware and software components. System engineering encompasses requirement definition, design, integration, and testing of all elements of a system, including hardware, software, interfaces, and interactions with other systems. It ensures that all parts of the system work together effectively and meet overall requirements.

4. System Design:
   This is a stage within system engineering focused on developing the system's architecture. It defines how the system's components will interact, what their structure will be, what resources are needed, and how integration and performance challenges will be addressed.

Some results in System engineering:

Conway's Law
Law of Requisite Variety
Emergence Awareness
Top-Down Design
CAP-theorem

Second-system effect
Design patterns
Design systems (system of components)
Single point of failure
State management (single store or not)

The foundations of system engineering and design involve several critical aspects that must be analyzed to build an effective and resilient system. First, it's important to identify whether the system has bottlenecks — components that limit performance and could become critical under increased load. Also, check for any SPOFs (Single Points of Failure) — elements whose failure could bring down the entire system.

Next, consider the structure: Is it modular, with independent components, or monolithic, with tightly integrated parts? Another key consideration is whether the system is distributed, with data and processes spread across multiple nodes, or centralized, where everything is located in one place.

For distributed systems specifically, the CAP theorem (Consistency, Availability, Partition Tolerance) must be applied, as it determines which of the three properties the system prioritizes. The Law of Requisite Variety is also vital: the system must have enough internal variety to respond adequately to environmental changes.

Equally important is analyzing the data flow within the system to detect any latency and assess the system’s ability to handle high data volumes. Lastly, define a Single Source of Truth to ensure consistency and accuracy of data across all system parts, preventing conflicts and errors.

These aspects enable the creation of a balanced, scalable, and fault-tolerant system that can handle heavy workloads and adapt to environmental changes.

In system design, **horizontal and vertical scaling** are crucial for ensuring performance and scalability:

- Vertical scaling involves increasing the capacity of a single server or machine by adding more resources like CPU, memory, or storage. However, this approach has limits—it gets harder and more expensive over time and can lead to bottlenecks that are difficult to fix.

- Horizontal scaling means adding more servers or nodes to the system, allowing load distribution across more machines. This significantly boosts system capacity without the limitations of vertical scaling. However, it requires more complex infrastructure for management, synchronization, and reliability—such as load balancing, fault tolerance, and data consistency across nodes.

Ultimately, the choice between horizontal and vertical scaling depends on factors like performance needs, budget, and infrastructure complexity. Horizontal scaling is typically more suitable for large, distributed systems due to its flexibility and growth potential with minimal limitations.

Component systems and design systems help build applications consistently and reliably.
Component systems reduce development time by providing ready-to-use components, so developers don't have to "reinvent the wheel" every time.
They also make teamwork between developers easier by offering a shared structure and standards.

Components are often designed to be flexible, supporting configurations such as themes and other customizations to adapt to different design requirements.
Additionally, design systems improve collaboration among developers by providing a shared structure, clear guidelines, and a unified visual language across the project.

Component systems and design systems ensure consistency and reliability in application development.
They provide a library of standardized, reusable components.

Component systems and design systems address the deeper challenge of not having to continually develop individual components from scratch.
Rather than focusing on ensuring that components don't conflict or duplicate one another, the goal is to have pre-designed, well-documented, ready-to-use components available. This eliminates the need for ongoing development consent or decisions at each stage of the project.
These systems ensure that components, such as themes or other configurations, are already aligned with the overall design principles, allowing for quicker development without compromising consistency.
By using a shared set of standards and guidelines, teams can focus on higher-level tasks and maintain a coherent and reliable application structure.

Component systems and design systems are essential tools for creating applications in a consistent and reliable manner.
By offering a collection of pre-built, standardized components, they significantly reduce development time.
Moreover, they facilitate collaboration among developers by providing a common framework and clear guidelines, ensuring that teams can work together more efficiently and maintain a unified approach throughout the project.

The **component-based approach** is a way of building frontend applications by breaking the user interface into small, reusable pieces called **components**. Each component is like a self-contained building block that includes its own **structure (HTML or template), behavior (JavaScript logic), and styles (CSS)**, all managed together in one place. These components can then be combined or nested inside one another to create complex interfaces.

This approach makes development more organized and scalable because each component handles a specific part of the UI independently. Components can manage their own **state** (dynamic data) and respond to user interactions without affecting other parts of the app. Because of this isolation, components are easier to develop, test, and maintain. Plus, their reusability means you can use the same component multiple times across your app, reducing duplicated code.

Example of graphics with component-based approach (Konva.js)

```
<div id="container"></div>

<script>
  var stage = new Konva.Stage({
    container: 'container',
    width: 500,
    height: 400,
  });

  var layer = new Konva.Layer();
  stage.add(layer);

  // Create a group
  var group = new Konva.Group({
    x: 100,
    y: 100,
    draggable: true,
  });

  // Add children to the group (circle and rectangle)
  var circle = new Konva.Circle({
    x: 0,
    y: 0,
    radius: 40,
    fill: 'green',
    stroke: 'black',
    strokeWidth: 2,
  });

  var rect = new Konva.Rect({
    x: 80,
    y: 20,
    width: 100,
    height: 60,
    fill: 'blue',
    stroke: 'black',
    strokeWidth: 2,
  });

  group.add(circle);
  group.add(rect);

  layer.add(group);
  layer.draw();
</script>
```

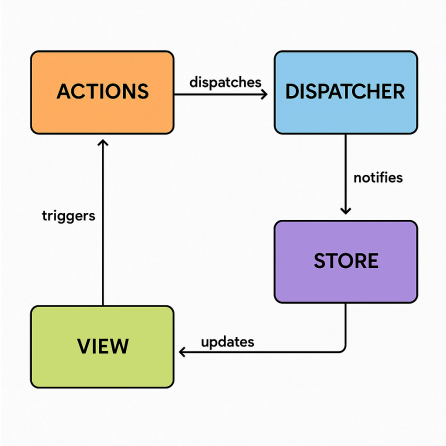
There are several software design patterns that help organize code and solve common problems. Patterns like PubSub (Publisher-Subscriber) allow components to communicate without being tightly coupled, where publishers send messages and subscribers listen for them. This decouples components, making the system more flexible. Another popular pattern is MVC (Model-View-Controller), which separates an application into three core parts: the Model (data and business logic), the View (user interface), and the Controller (input handler). This separation of concerns improves maintainability and scalability in applications.

The Flux pattern (often associated with Redux in JavaScript applications) is designed for predictable state management. It uses a unidirectional data flow where actions trigger changes to the state stored in a store, and views update in response to those changes. This pattern helps maintain consistency in large, complex applications. It's particularly useful for managing

state in UI-based applications like React, where having a central state ensures better debugging and state control.

In Flux, data flows in a single direction: user interactions trigger actions, which are dispatched to a dispatcher; the dispatcher then notifies the relevant stores that hold application state and logic. Once a store updates its data, views (typically React components) re-render accordingly. This structure enforces a predictable flow of data, making it easier to manage and debug complex UI states compared to traditional two-way data binding systems.

Stores are containers for application state and business logic. Unlike MVC's models, stores don't just hold data—they also handle logic for how data should respond to actions. They listen for dispatcher events and update their state accordingly. Stores also emit a change event, signaling views to re-render if necessary.



GoF design patterns are important in software design because they provide proven, reusable solutions to common design problems. They help developers create code that is modular, flexible, and easier to maintain or extend, especially in large or complex systems. Understanding these patterns also improves communication among developers, as they offer a shared vocabulary for discussing design ideas.
The GoF (Gang of Four) patterns, outlined in the seminal book "Design Patterns: Elements of Reusable Object-Oriented Software", cover 23 fundamental design patterns, including the Observer, Singleton, and Factory patterns. These patterns focus on solving common object-oriented design problems, such as ensuring a class has only one instance (Singleton) or creating objects without specifying the exact class (Factory). The Observer pattern is useful for managing state changes and notifying dependent objects, often used in event-driven programming or GUI applications.
Creational patterns (like Singleton, Factory Method, and Builder) deal with object creation mechanisms. Structural patterns (such as Adapter, Decorator, and Facade) focus on composing

objects and classes into larger structures. Behavioral patterns (including Observer, Strategy, and Command) manage algorithms, communication, and responsibilities among objects. These patterns promote flexibility, reusability, and maintainability in software architecture.

Emergence Awareness in Systems Engineering

Emergence means:

The whole system shows behaviors or properties that its individual parts do not have on their own.

In short:

The system is more than just the sum of its parts.

Some important properties appear only when parts interact together.

The CAP theorem states that in any distributed system, it is impossible to simultaneously guarantee Consistency, Availability, and Partition tolerance. Consistency means that all nodes see the same data at the same time, Availability ensures every request receives a response, and Partition tolerance means the system keeps working even if network failures split it into parts. Since partitions can always happen in a real network, the system must sacrifice either consistency or availability during a partition.

When a partition occurs, a system choosing consistency will reject some requests to keep data correct and synchronized across nodes. Meanwhile, a system choosing availability will respond to all requests even if the data might not be fully up-to-date. Different distributed databases and services make different trade-offs depending on their needs — for example, some prioritize fast user responses (favoring availability), while others prioritize always-correct data (favoring consistency).

In practice, designers use the CAP theorem to decide what to prioritize based on the system’s purpose. For highly critical systems like banking, consistency is more important, while for social media or shopping websites, availability might be prioritized. The CAP theorem doesn't mean systems can never be consistent and available — it means that under a network partition, you can guarantee only two out of the three properties.

Eventual consistency sacrifices immediate accuracy for high availability, and it's a common strategy in distributed systems constrained by the CAP theorem. It's used in AP systems (like DynamoDB, Cassandra) to favor availability and partition tolerance over immediate consistency.

AP systems are distributed systems that prioritize:

A — Availability: Every request receives a response (even if it’s not the latest data).
P — Partition Tolerance.

Eventual consistency is a consistency model used in distributed systems (like databases or caches) where updates to data are not immediately visible across all nodes, but over time, all nodes will become consistent.

In other words:
- When you update data in one place, other parts of the system might temporarily see an older version.
- But the system guarantees that eventually everyone will see the same data.

This model is useful when availability and scalability are more important than instant consistency.
Example: large services like Amazon or social networks often rely on eventual consistency.

The second-system effect refers to a phenomenon where, after a successful initial system (or product), the next version or iteration of that system becomes overly complex and over-engineered. This often happens because the developers or creators, emboldened by the success of the first system, attempt to add more features, functionalities, or improvements, thinking they can handle everything that was left out before. However, this leads to a system that is more complicated and harder to manage than the original.

The concept was introduced by Fred Brooks in his book “The Mythical Man-Month”, where he discusses how engineers and developers tend to overcompensate for perceived shortcomings in their first design when working on the second one. This can result in a product that is difficult to maintain, more prone to bugs, and harder for users to understand and use.

YAGNI (You Aren't Gonna Need It) is related to the second-system effect in that both address the risks of overengineering and unnecessary complexity.

Second-system effect happens when the success of an initial system leads to overcomplication in the second version, where developers add too many features and make the system more complex than it needs to be.

YAGNI prevents this by promoting simplicity and discouraging the inclusion of features that aren’t absolutely necessary at the moment.

In essence, YAGNI can be seen as a safeguard against the second-system effect, helping developers avoid the temptation to over-engineer a system by reminding them not to add features unless they are genuinely required. By following YAGNI, developers are more likely to keep things simple and avoid the overcomplexity that often emerges with the second iteration of a product.

YAGNI stands for "You Aren't Gonna Need It." It is a principle in software development that advises against adding functionality or features to a system until they are absolutely necessary. The idea is to avoid overengineering or prematurely implementing features that may never be used, which can lead to wasted time, increased complexity, and maintenance challenges.

The YAGNI principle encourages developers to focus only on the immediate needs of the project and to avoid anticipating future requirements that may never materialize. By doing so, it helps keep the codebase simple, flexible, and easier to maintain.

In practice, YAGNI means:
- Building only what is required for the current task.
- Avoiding speculative or unnecessary features.
- Reducing the risk of creating unnecessary technical debt.

This principle is commonly associated with Extreme Programming (XP) and Agile methodologies.

Conway's law describes the link between communication structure of organizations and the systems they design.

Conway’s Law says:

"Any organization that designs a system will produce a design whose structure is a copy of the organization's communication structure."
— Melvin Conway, 1967

What it really means

- If your teams are organized in separate groups that don’t talk much, the system they build will also have separate, poorly integrated parts.
- If your organization is highly collaborative and cross-functional, the system will likely be better integrated and more cohesive.

Simple Example

Suppose:
- A company has three separate teams:
  - Team A (Frontend)
  - Team B (Backend)

- Team C (Database)

Each team mostly talks internally but not much across teams.

Result:
The product they build will naturally have a frontend, backend, and database with strong separation — and possibly poor communication between these parts (e.g., bad APIs, slow data sync).

Important consequences
- Organizational problems → System problems
  If teams don't communicate, you get bugs, integration issues, and fragile systems.

- You can't design a system independently of the organization.
  Changing the system's architecture requires changing the team structure too!

- Deliberate team design is important.
  Modern practices like DevOps and agile cross-functional teams are partially about fixing the Conway's Law trap — making communication patterns match desired system structure.

In system engineering, the **Law of Requisite Variety** states that for a system to effectively manage external disturbances, its internal complexity (or variety) must be at least equal to the variety of those disturbances. For example, consider the design of a spacecraft intended to operate in space or on another planet. The external environment poses a multitude of challenges, such as extreme temperature fluctuations, radiation, micrometeoroid impacts, and communication delays. Each of these factors represents a disturbance with its own complexity or variety. If the spacecraft's internal systems were too simple or lacked the ability to handle these disturbances, it would fail when faced with unexpected conditions. The disturbance variety V_D is high, and without an adequately complex response mechanism, the spacecraft would not function effectively in the unpredictable environment of space.

To meet this challenge, engineers increase the spacecraft's response variety V_C by incorporating redundant systems, autonomous fault detection and correction, reconfigurable hardware, and advanced thermal regulation systems that can adjust based on internal and external conditions. These features ensure that the spacecraft can handle a variety of disturbances — from mechanical malfunctions to drastic temperature changes — without compromising its mission. By applying the Law of Requisite Variety, the system becomes resilient, as it can adapt to unforeseen circumstances and continue to operate smoothly. This principle is crucial in engineering fields such as robotics, aerospace, and software design, where maintaining control and stability in the face of varying conditions is essential.

The disturbance variety V_D is huge.
If the spacecraft were designed simply, with no backups and no flexibility, any single unexpected event could cause total mission failure.

Solution — Applying the Law of Requisite Variety:
Engineers increase the system's response variety V_C by:

Adding redundant systems (two or three copies of critical parts like computers, power supplies, communications)

Designing autonomous fault detection and correction software

Building thermal regulation systems that can switch modes based on internal conditions

Using reconfigurable hardware (for example, using different pathways if one is damaged)

The Law of Requisite Variety, formulated by W. Ross Ashby, states that a controller must possess at least as much variety as the disturbances it aims to manage. In technical terms, "variety" measures the number of possible states or behaviors, often quantified using Shannon's entropy. When the environment or system being regulated has high variability, a controller with limited responses will fail to maintain desired conditions. Thus, control and stability require the regulator’s flexibility to match or exceed the complexity of the disturbances.

Consider a thermostat as a practical technical example. The thermostat’s task is to maintain room temperature at 22°C. If the environment changes — with cooling winds, heating sun, or humidity shifts — the disturbances introduce multiple possible temperature changes. A simple thermostat with only two actions — "heater ON" and "heater OFF" — represents low response variety. If the environment only cools, heating is sufficient; but if the environment also overheats the room, the thermostat cannot cool it, revealing that the controller's variety is insufficient to cover the disturbance variety.

Formally, if we denote the disturbance variety as V_D and the controller variety as V_C, the law demands $V_C \geq V_D$ for effective regulation. In cases where $V_C < V_D$, full control is impossible: there will always be some external states that the controller cannot adequately respond to. This constraint is deeply tied to information theory: disturbances with higher entropy require regulators capable of producing responses with equal or higher entropy to absorb the unpredictability.

To solve the thermostat problem, the controller's variety must be increased — for example, by adding both a heater and an air conditioner. Now, the system can react to both cooling and heating disturbances. As a result, V_C is increased, satisfying the Law of Requisite Variety. More generally, complex systems (biological, mechanical, organizational) must either expand their response set or limit the incoming disturbances to remain effective.

In broader systems — like robots adapting to dynamic environments, immune systems responding to a range of pathogens, or businesses facing diverse customer needs — the Law of Requisite Variety underpins the necessity of flexibility. A rigid, low-variety system inevitably fails when facing complex, high-variety challenges. Therefore, understanding and designing for requisite variety is essential in all fields where control, adaptation, and survival under change are critical.

In system engineering, **Kerckhoffs's principle** emphasizes that a system should be secure even if everything about it, except the key, is public knowledge.

Originally formulated in the context of cryptography by Auguste Kerckhoffs in the 19th century, this principle has broader applications in system engineering, particularly in the design of secure, reliable, and maintainable systems. Here's how it's interpreted in system engineering:

Core Idea:
Security through obscurity is not a reliable strategy.

In System Engineering Context:
1. Design Transparency: System security should not rely on the secrecy of algorithms, architectures, or implementations. Only the keys or passwords (small secrets) need to be kept confidential.

2. Modular Replacement: Systems should be designed so that components (e.g., a compromised algorithm) can be easily swapped or upgraded without redesigning the whole system.

3. Auditability & Testing: Open designs are easier to test, audit, and verify, improving both reliability and trustworthiness.

4. Resilience to Breach: If a part of the system (except the key) becomes known, the rest of the system should not be compromised.

Practical Examples:
- Encryption algorithms like AES are fully public, but their strength lies in the secrecy of the key.
- Secure software systems publish their source code (e.g., open-source libraries), relying on strong key management and secure practices rather than secrecy of code.

The Kerckhoffs's principle was formulated by cryptographer Auguste Kerckhoffs in his work “Military Cryptography” (1883). He described this principle in the context of military cryptography, emphasizing that the security of encryption should rely solely on the secrecy of the key, not on the secrecy of the encryption algorithm itself. This principle became the foundation for many modern cryptographic systems and ensured their resilience against potential attacks, even if the algorithm becomes public.

This means that even if an adversary has knowledge of algorithms like RSA and AES, we can still safely use them, because their designers took the Kerckhoffs's principle into account.

This concept is widely embraced by cryptographers, in contrast to security through obscurity, which is not.
This approach relies on the principle of hiding something in plain sight, akin to a magician's sleight of hand or the use of camouflage. It diverges from traditional security methods, such as physical locks, and is more about obscuring information or characteristics to deter potential threats.
Security by obscurity alone is discouraged and not recommended by standards bodies.

Cybersecurity is the protection of the vital interests of individuals, society, and the state while using cyberspace.
Cyberspace is an environment (virtual space) that provides opportunities for communication and/or the realization of social relations using the Internet and/or other global data transmission networks.
Cybersecurity is guaranteed through the implementation of multilayered technical protection mechanisms, which include data encryption (AES, RSA), the use of secure information transmission protocols (TLS, HTTPS), and access control systems (RBAC, OAuth).
Tools such as firewalls, intrusion prevention systems (IPS), and antivirus solutions actively protect the network perimeter and internal resources. Monitoring systems (SIEM) analyze events in real-time, detecting anomalies that may indicate an attempted attack.
A key aspect is the implementation of modern authentication methods, such as biometrics or two-factor authentication (2FA), as well as control over software updates to eliminate vulnerabilities. The use of container isolation technologies (Docker, Kubernetes) and virtualization (Hyper-V) limits the impact zone in case of compromise. Regular penetration testing and security audits help identify weaknesses and improve overall resilience to cyber threats.
Continuous creation of backups of critical data and the ability to quickly restore it in the event of a cyberattack or technical failures ensures the resilience of the information system.
Cybersecurity is ensured through the use of cryptographic algorithms that protect data during transmission and storage. Symmetric and asymmetric encryption algorithms, such as AES and RSA, ensure the confidentiality and integrity of information.
At the same time, cryptanalysis is used to test the strength of ciphers, helping to identify potential weaknesses in protection. Combined with steganography, which hides data inside other objects (images, audio files), this creates an additional level of protection, making unauthorized access to information more difficult.
The use of CAPTCHA in forms blocks automated attacks, such as DDoS and brute-force, preventing unauthorized access to systems. Data validation and sanitization help prevent injections, such as SQL injection, which can lead to leaks or damage to confidential information.
Transmitting passwords using the POST method instead of GET eliminates the risk of them being stored in cache or logs, which is critical for maintaining confidentiality.

Following password storage rules, including hashing and avoiding the use of cookies, protects against data theft in the event of leaks or XSS-type attacks. These practices enhance resilience to threats, maintaining the integrity and security of military systems, which are the foundation of effective resource and data management in the face of modern cyber threats.

VPNs (Virtual Private Networks) add an additional layer of security by encrypting internet traffic and masking the user's IP address, ensuring secure connections to remote networks. VPNs help protect sensitive data when accessing public or untrusted networks by creating a private tunnel for data transmission. This is especially important for safeguarding the privacy of individuals and organizations, ensuring that their online activities are protected from hackers and surveillance.

In addition to common defenses, preventing specific attacks like Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), Man-in-the-Middle (MITM), Downgrade, Collision, Denial-of-Service (DoS), Slowloris, Timing, Lucky Thirteen, Padding Oracle, Terrapin, Oracle, KRACK, and Frequency Analysis requires a layered approach. Measures such as input sanitization, secure communication protocols, anti-CSRF tokens, SSL/TLS encryption, constant-time algorithms, updated cryptographic practices, and rate limiting are essential in thwarting these attacks. Input sanitization ensures that user-provided data is cleansed and checked before processing, preventing malicious code injection, a common cause of XSS. Secure communication protocols, like SSL/TLS, encrypt data in transit to protect against MITM attacks, where adversaries intercept or alter communications. Anti-CSRF tokens prevent unauthorized requests by ensuring actions on a website are initiated by legitimate users. Constant-time algorithms are designed to prevent timing attacks, where attackers try to infer sensitive information based on response time. Using secure, up-to-date cryptographic algorithms such as AES and RSA and ensuring proper key management mitigates risks like frequency analysis and brute-force decryption. Rate limiting restricts the number of requests a user can make within a specific timeframe, helping to prevent DoS and DDoS attacks. Strict Transport Security (HSTS) ensures browsers always connect using secure HTTPS connections, defending against SSL/TLS downgrade attacks and cookie hijacking. Preventing downgrade attacks also involves configuring servers to support only modern, secure SSL/TLS versions. Protecting against Padding Oracle Attacks requires using authenticated encryption methods with secure key management. Finally, addressing KRACK vulnerabilities in Wi-Fi networks requires keeping encryption protocols up to date, such as transitioning to WPA3 and applying necessary patches. A layered defense strategy, where each layer strengthens security, reduces the likelihood of successful attacks, and ensures a more resilient system, especially when combined with regular audits and well-trained security teams.

Defence in Depth in Your Application

Defence (or security) in depth is a concept in cybersecurity and software development that draws an analogy with military front lines. On the battlefield, there are several layers of defense so that if one is breached, another can hold the line.

This approach may seem similar to the principle of eliminating SPOF (Single Point of Failure), but that's a misconception. The idea is not to assume that our encryption or protection will inevitably fail. Rather, we should consider that if the primary defense layer falls—say, an algorithm gets cracked—then the rest might collapse as well. In the case of SPOF, we focus on the likelihood of a single system component failing, like a server. If a top (powerful) encryption algorithm is bypassed, everything else becomes useless. However, if a server fails, it makes sense to have a backup.

Defence in depth makes sense when your security measures don't guarantee complete protection. For example, installing multiple locks on a door along with a surveillance camera adds different levels of security.

This approach is also justified when the protection layers operate across different dimensions. For instance, two-factor authentication doesn't assume your password has been cracked—it protects in case someone has learned or stolen it.

SPOF

To build reliable systems, it is essential to eliminate SPOF (Single Point of Failure). This means that ideally, there should be no single point or component in the system whose failure would cause the entire system to go down. In other words, for the system to fail, at least two independent modules must break down. These modules should be positioned in such a way that the likelihood of simultaneous failure is minimized.

Various methods are used to eliminate SPOF, including:

- Modularity and isolation.
  The system is divided into independent modules that are isolated from each other and interact through clearly defined interfaces. Isolation allows each module to be tested separately, ensuring that its functionality does not depend on other parts of the system beyond the interface.

- Closed to modification.
  Modules on which many other components depend should be closed to changes, since altering them could cause system failures. This aligns with the "Open/Closed" principle from SOLID, but it's important to note that while being closed to modification increases safety, openness to extension can be risky. Extending a module may lead to unexpected errors.

- Access rights.
  Access rights between modules and different systems must be controlled and follow the principle of Least Privilege, which means granting only the minimum rights necessary to perform a task. This reduces the risk of unauthorized changes and errors.

- Unit testing/Auto testing.
  Automated testing, especially unit testing, improves system reliability and reduces feedback time between the tester and the system. It enables early detection of bugs in individual modules and helps maintain system stability.

- Supervisor, Watchdog timer.
  The system should include components that continuously monitor its operation and attempt to automatically restart it in case of a failure. This helps ensure uninterrupted operation in critical situations.

Therefore, eliminating SPOF and applying these practices help create more reliable and fault-tolerant systems.

System design and engineering are guided by principles aimed at creating efficient, scalable, and reliable systems. Key principles include modularity, where systems are broken down into manageable components, and abstraction, which simplifies complex systems by hiding details behind clear interfaces. Additionally, scalability is essential to handle increasing workloads, while fault tolerance ensures that the system remains operational even when individual components fail. Security, efficiency, and maintainability must also be prioritized, ensuring that the system is not only functional but also sustainable and easy to manage over time.

A crucial aspect of system design is flexibility, allowing systems to evolve in response to new requirements or technologies. Design patterns such as service-oriented architecture (SOA) and event-driven architecture promote loose coupling and adaptability, enabling different components to interact efficiently. The principle of **separation of concerns** ensures that different system layers, such as data processing and user interfaces, are isolated to reduce complexity and improve maintainability. Redundancy and failover mechanisms are essential to ensure high availability and resilience, while monitoring and feedback loops allow for continuous improvement based on real-world performance.

Finally, ethical considerations, such as user privacy and data protection, are integral to modern system design. Systems should comply with legal and regulatory standards, ensuring that they operate within the bounds of applicable laws and industry standards. Environmental sustainability is also becoming an important factor in system design, with a focus on minimizing the carbon footprint and optimizing resource consumption. By balancing technical excellence with ethical and operational concerns, system designers can create solutions that are not only effective but also responsible and adaptable to future challenges.

Ensuring Reliability

To build a reliable system that operates faultlessly under high security and stability demands, it is essential to follow a set of key principles and practices. Here's what this may look like in the context of system development and operation:

0. Clear Definition of Operating Conditions
First, it is necessary to understand the environment in which the system will operate. This includes both software and physical environments. For critical infrastructure or systems in high-stakes domains (healthcare, energy, transportation), these conditions must be clearly described. For example, whether the system will function under resource constraints, high temperatures, or strict latency requirements. Such conditions influence architecture and tool selection. Standards such as MIL-STD-810 and NIST CSF can guide this process.

1. Provide Hardware Status Checking Mechanisms: Indicators, Watchdog Timer, etc.
A Watchdog Timer (WDT) is a mechanism (device) used to monitor the operation of computer systems or microcontrollers. Its primary task is to prevent system hangs by automatically initiating a reboot or another action when a fault is detected. If the program fails to reset the timer regularly (due to a crash or freeze), the WDT triggers corrective action, improving system reliability—crucial in fields like medicine, defense, or aviation.

2. Use of RISC Processors
Selecting the right hardware is crucial. RISC (Reduced Instruction Set Computing) processors use a simplified set of instructions, each executed in a single clock cycle. This enables high performance, low power consumption, and predictable system behavior. Their simplicity makes them ideal for embedded systems where stability, efficiency, and reliability are essential.

3. Use of Microkernel-Based Operating Systems
To enhance safety and stability, it is crucial to use microkernel-based operating systems like QNX or Minix. Microkernels separate system components into isolated modules, reducing the likelihood that an error in one will crash the entire system. This modularity improves error handling and resilience.

4. Isolation from Third-Party Software and Factors
Isolation is a key aspect of security and reliability. The system should limit access to third-party software and components through access restrictions, containerization, or virtualization. Avoid unverified libraries or drivers unless proven reliable and secure.

5. Selection of Reliable Drivers and Software
Unreliable drivers or programs can lead to major issues. Choose only tested and certified components. Regularly update and audit them for vulnerabilities.

6. Restricting Application Access
Access to system resources and data must be tightly controlled using OS-level access rights and control mechanisms. Apply the **Principle of Least Privilege**—each program should have only the permissions it needs.

7. Testing at Every Stage and Level
Reliable systems require testing at every development stage—unit, integration, and system testing. Automated testing should be a built-in part of the development process. Each component must be tested in isolation before integration.

8. Automated Tests and BIT (Built-In Testing)
Implement automated testing at all levels: unit, integration, stress, reliability, and security testing. BIT allows testing during runtime without stopping the system or adding load.

9. Logging and Monitoring
Logging is essential for identifying and fixing issues. Logs should be collected at all stages and integrated with automated monitoring for timely response and system improvement.

10. Backup and System Recovery
The system must support data backups and fast recovery. It should also support failover, switching to backup systems in case of component failure. Avoid Single Points of Failure (SPOF) by duplicating critical components (servers, power supplies, networks).

11. “Daemons” and Supervisors for Process Recovery
Ensure continuous operation by using daemons to monitor and restore failed processes. Use supervisors to manage program behavior and decide when recovery is needed. The term “daemon” originates from MIT’s MAC project, inspired by Maxwell’s Demon—an entity working in the background to manage order.

12. Code Review and Thorough Manual Testing
Each new feature must undergo code review to catch potential bugs or vulnerabilities. Manual testing is crucial for critical components to find issues that automated tests might miss.

13. Onion Architecture
The system should follow clean (onion) architecture, where lower layers are "frozen," and extensions occur at the top layers. This minimizes the need for full regression after each change. Instead, new features are added via specialized modules, reducing the risk of bugs.

Clean architecture ensures clear separation of responsibilities, keeping business logic isolated from implementation details (e.g., database or services). This reduces component coupling and improves testability and maintainability.

14. Using Reliable Programming Languages

It is recommended to use a compiled language with strict typing and no dynamic arrays to avoid memory leak issues. Restrictions on Garbage collector.

For developing reliable systems, it is important to choose programming languages that ensure security and efficiency at all stages. The most suitable languages are those that provide high performance and resource control, such as C, C++, Rust, Ada, Haskell, Erlang, Elixir, Fortran, and possibly Python. The choice of language should depend on the specific requirements of the system.

15. Network

When discussing network-connected systems, it is essential to consider aspects such as protocols that ensure data integrity, mechanisms for server failure recovery, caching, firewall configurations, and other security and optimization measures. The CAP theorem should also be considered.

Typically, a load balancer and auto scaling can be deployed to ensure high availability of a server cluster at the system level.

16. The System Should Rely on Mathematically Proven Methods: Correct Algorithms, Theorems, etc.

Mathematical techniques, such as algorithm complexity theory (time and space analysis), should be used to assess the effectiveness and correctness of algorithms. Algorithms must be proven through mathematical induction or other methods to guarantee their correctness and stability.

For example, Claude Shannon proved several theorems about perfectly secure ciphers, channel bandwidth, and the sampling theorem.

17. Circuit Breaker Pattern and Similar Patterns

The Circuit Breaker pattern is applied to prevent cascading failures in distributed systems. Its essence lies in automatically stopping requests to a service that has started returning errors frequently. Instead of repeatedly attempting failed requests and unnecessarily overloading the system, the Circuit Breaker temporarily "breaks the chain," instantly returning an error.

The mechanism has three states: Closed — requests pass normally, Open — requests are blocked after a threshold of errors is reached, and Half-Open — a test mode where it checks if the service has recovered. If test requests succeed, the system returns to normal mode (Closed); otherwise, it opens the circuit again (Open).

For example, Service A calls Service B to process payments. If B starts failing frequently, the Circuit Breaker in A stops requests to B for a certain period, maintaining A's stability. After the timeout, A retries, and only if everything is fine does it resume working with B.

Service B is temporarily unavailable (e.g., due to technical maintenance or overload).

Service A continues to send requests, each of which:
- waits for a response for a long time (timeout),
- consumes resources (CPU, threads),
- and ultimately fails.
This reduces A's performance and may lead to a complete system failure.
Solution:
Use Circuit Breaker in Service A when calling Service B.

18. Hamming Codes and Error-Correcting Memory

ECC (Error-Correcting Code) memory is used in most computers where data corruption cannot be tolerated under any circumstances, such as for scientific or financial computations. ECC keeps the system's memory error-free by ensuring that the bits read from each word always match the bits originally written, even if one or more bits have been flipped to an incorrect state for various reasons. Some types of memory (without ECC) with parity control allow errors to be detected but not corrected, while in other cases, errors are not detected at all.

Hamming code is used in some applications in the field of data storage, particularly in RAID 2; moreover, the Hamming method has long been applied in ECC memory and allows "on-the-fly" correction of single-bit errors and detection of double-bit errors.

Hamming code is a code that automatically detects errors that occur during transmission. Suppose we have a device (e.g., a computer) that processes programs in binary code, so its program consists of a chain of binary code, such as 8 binary digits (8 bits), like:
10100010, 00010101, 00000001, 10101010.
During reading or transmission of such code, an error may occur, causing several bits to change, i.e., bits in binary words. A binary word in such cases consists of eight bits, like 10100010. Hamming code allows us to fully correct errors in the code if no more than one bit is changed in each word (i.e., no more than one digit in an 8-bit sequence).

The essence of the Hamming code is to assign a control bit to each word, which we mark with a dash:
10100000 | 0,
00010101 | 1,
00000001 | 1,
10101010 | 0,
The control bit is set in such a way that the total number of ones in the word is even or zero. Thus, we know that each word must contain an even number of ones or zero, otherwise, the code is corrupted. If we equip our computer with a bit counter that counts bits modulo two, and when the modulo two result is zero, the program continues reading; otherwise, an error message will appear to correct the code.

Example of a "broken" Hamming code, i.e., corrupted:
10101000 | 0, - error, the number of ones is not even.

00010101 | 1,
00000001 | 0, - error, the number of ones is not even.
10101010 | 0.

You can use multiple control bits to track more than one error, for example, by adding two control bits. The first will complement (and check) one half of the binary word, and the second will check the other half of the word.
Hamming code with two control bits:
10001010 | 10 (which means 1000|1 and 1010|0).
“The purpose of computing is insight, not numbers” (Richard Hamming, “Numerical Methods for Scientists and Engineers”).

Conclusion

A reliable system is the result of careful planning, strict quality and security control at all stages of development and operation. It is important to use the right tools and practices, such as microkernel operating systems, test automation, backup and monitoring, as well as using reliable programming languages and systems. Only this way can we create a system that is resilient to failures and attacks and ensures a high level of reliability over long periods of operation.

It is always worth evaluating the damage in case of a system failure and planning adequate security measures.

Memory reliability is ensured through several technical and organizational approaches that reduce the likelihood of errors and data loss:

1. Error Correction (ECC — Error-Correcting Code):
Special codes are added to each data block to detect and correct errors that may occur during storage or transmission. For example, ECC memory can automatically correct single-bit errors.

2. Redundancy:
Critical data is stored in multiple copies (mirroring, RAID arrays) so that if one copy is lost or damaged, another can be used.

3. Power Failure Protection:
The use of uninterruptible power supplies (UPS) and capacitors preserves information in the event of sudden power outages.

4. Regular Data Scrubbing:
Memory is periodically scanned for errors with automatic correction of any issues found.

5. Temperature and Electrical Control:

Reduces the likelihood of failures by monitoring temperature, humidity, and power supply voltage.

6. Software Backups:
Regular creation of data copies on external or remote media (local disks, cloud services).

7. Use of Reliable Memory Types:
For instance, servers often use ECC RAM instead of regular RAM, and critical applications rely on specialized flash memory with a longer lifespan.

Error correction code memory (ECC RAM) is a type of computer data storage that uses an error correction code to detect and correct n-bit data corruption which occurs in memory.

The most-common error correcting code, a single-error correction and double-error detection (SECDED) Hamming code, allows a single-bit error to be corrected and (in the usual configuration, with an extra parity bit) double-bit errors to be detected. Chipkill ECC is a more effective version that also corrects for multiple bit errors, including the loss of an entire memory chip.

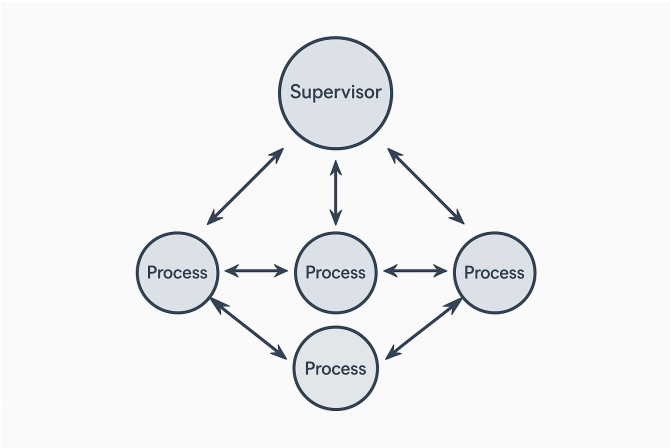
Chipkill is IBM's trademark for a form of advanced error checking and correcting (ECC) computer memory technology that protects memory systems from single memory chip failures and multi-bit errors from any portion of a single memory chip. One simple scheme to perform this function scatters the bits of a Hamming code ECC word across multiple memory chips, such that the failure of any single memory chip will affect only one ECC bit per word. This allows memory contents to be reconstructed despite the complete failure of one chip. Typical implementations use more advanced codes, such as a BCH code, that can correct multiple bits with less overhead.

Process supervising

In Elixir programming language, a process is a lightweight, independent unit of execution that can run code, hold its own state, and communicate with other processes through messages. It’s not like an operating system (OS) process — Elixir processes are much smaller, faster to create, and you can have thousands or even millions of them running at the same time.

Each process:
- Has its own memory (it doesn’t share data with others directly).
- Runs concurrently with other processes (like actors on different parts of a stage).
- Communicates safely using message passing (no shared variables to fight over).

Under the hood, Elixir processes are managed by the Erlang VM (BEAM), which is specially designed to make these tiny processes extremely efficient and fault-tolerant.



When a process in Elixir crashes, it doesn’t bring down the whole system because each process is isolated. Instead, the crash is immediately reported to its supervisor. The supervisor’s job is to decide what to do next — it can restart just the crashed process, restart a group of related processes, or even ignore it if the failure isn’t critical. This way, problems stay contained, and the overall system remains stable and responsive without needing manual intervention.

This strategy is known as “let it crash.” Instead of trying to prevent every possible error inside each process, Elixir systems are designed to detect errors quickly and recover automatically. Supervisors use simple rules, like restarting a failed actor in a play or replacing a cook in a kitchen, to keep everything running smoothly. This approach is what makes Elixir so powerful for building fault-tolerant, resilient systems.

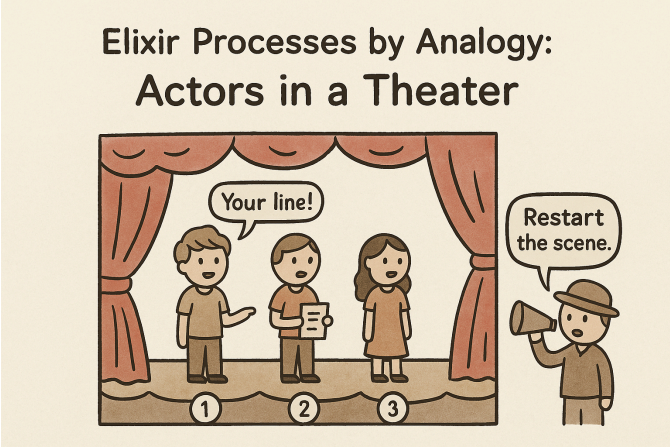
Imagine a kitchen with many chefs.
Each chef is an Elixir process.

- Every chef has their own task (one is chopping, one is frying, one is baking).
- They work independently. One chef doesn't need to know how another chef is doing their job — only that they're doing it.
- If one chef burns a dish (crashes), it doesn’t burn down the whole kitchen. Maybe a supervisor (head chef) notices and says: “Start over” or “Get a new chef.”
- Chefs can send each other notes (messages): “Hey, when you're done chopping, send me the carrots.”
- The kitchen is still one team — but the magic is that every chef can be doing their work without blocking the others.

In Elixir:
- A process is lightweight — like a chef focused on a small, specific job.
- Messaging is safe — they don’t mess with each other's tools or workspace.

- Failure is okay — there’s a strategy (supervisors) for handling mistakes without collapsing the kitchen.

Also imagine a big stage play.
Each actor is an Elixir process.



- Every actor has their own script (what to say, when to move, etc.).
- Actors don't share brains — one actor cannot directly control another. If they need something, they send a message (“Your line!” or “Pass me the prop!”).
- If an actor forgets a line (crashes), a director (supervisor) can step in: maybe prompt them, send an understudy, or restart the scene.
- Actors work together to make the play happen, but each one operates separately.

In Elixir terms:
- A process is like an actor reading its script.
- Messages are how actors interact.
- Supervisors are like directors, ready to handle mistakes and keep the show running.

In both analogies — kitchen or theater — the important idea is:
Elixir processes are independent, talk through messages, and recover gracefully if something goes wrong.

What is the dining philosophers problem?

The dining philosophers problem is a classic example of a synchronization issue that arises in multitasking programs and aims to demonstrate problems related to resource contention. This problem is often used to illustrate the principles of multiprocessor programming and synchronization in environments where multiple threads or processes share common resources.

The scenario of the dining philosophers problem is as follows:
1. There are five philosophers sitting at a round table.
2. Each philosopher has a plate of food in front of them.
3. Between each pair of philosophers, there is a chopstick.
4. A philosopher can perform two actions: think and eat.
5. To eat, a philosopher must take two chopsticks (left and right) and then take the food.
6. After the meal, the philosopher puts the chopsticks back on the table.

The problem is how to synchronize the actions of the philosophers to avoid deadlock and starvation, where some philosophers might not get a chance to eat due to improper synchronization.
Various algorithms and approaches in programming can be used to solve the dining philosophers problem, including semaphores, locks, and other synchronization mechanisms. This problem helps to understand important concepts in synchronization and avoid issues related to simultaneous access to shared resources in multitasking systems.
The dining philosophers problem can be solved using semaphores or mutexes, which are tools for synchronizing access to resources in multitasking programs.
Here is a way to solve this problem using semaphores:
- Create an array of semaphores, where each array element represents a chopstick.
- Initialize all semaphores to the "1" state, which means all chopsticks are available.
- Each philosopher must use two semaphores: one for the left chopstick and another for the right chopstick.
- The philosopher must use "wait" and "signal" semaphore operations to access the chopsticks.

Pseudocode for a philosopher might look like this:

```
while True:
think()
take_left_chopstick()
take_right_chopstick()
eat()
put_right_chopstick()
put_left_chopstick()
```

The operations "take_left_chopstick", "take_right_chopstick", "put_left_chopstick", and "put_right_chopstick" are implemented using semaphores and are carried out using "wait" and "signal".

For example:

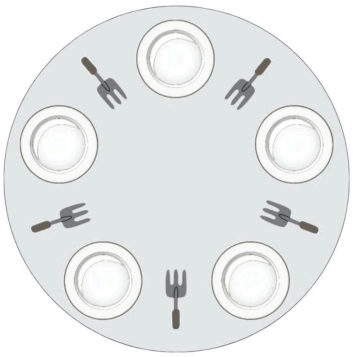
```
def take_left_chopstick():
```

```
left_chopstick_semaphore.acquire() def take_right_chopstick():
right_chopstick_semaphore.acquire() def put_left_chopstick():
left_chopstick_semaphore.release() def put_right_chopstick():
right_chopstick_semaphore.release()
```

This approach to solving the dining philosophers problem using semaphores ensures synchronization of access to the chopsticks and guarantees that philosophers will not simultaneously take the same chopstick, which could lead to deadlock.
A semaphore is a concept used to manage access to resources, especially in multitasking or multiprocessor environments. A semaphore is a variable or data structure that allows threads (or processes) to synchronize their work and interact with each other to avoid conflicts and ensure proper execution order.

There are two main types of semaphores:
1. Binary semaphore (mutex): This is a semaphore with two possible states—free and blocked. It is used when access to a resource must be allowed or denied to only one thread (process) at a time.
2. Counting semaphore: This type of semaphore allows access to a resource to be limited to a certain number of threads or processes. It maintains a counter that can be incremented or decremented by threads to control access to the resource.
Semaphores are commonly used in operating systems and when developing multitasking programs to manage concurrent access to resources such as files, printers, network connections, and so on. They are essential tools for ensuring correctness and reliability in the interaction of threads in programs and operating systems.

The dining philosophers problem.



In 1965, Dijkstra posed and solved the synchronization problem, which he called the "Dining Philosophers Problem."
Edsger W. Dijkstra (1930–2002) was a Dutch pioneer in the field of computer science.

The dining philosophers problem can be formulated as follows: Five philosophers are sitting at a round table. Each philosopher has a plate of spaghetti. The spaghetti is so slippery that the philosopher needs two forks to eat it. There is one fork between each pair of plates. For simplicity, we will assume that a philosopher's life consists of alternating periods of eating and thinking. When a philosopher becomes hungry, they try to pick up their left and right forks, one by one, in any order. If they manage to pick up both forks, they eat for a while, then put down the forks and continue thinking. The question is: can you write a program for each philosopher that does what it is supposed to do and never gets stuck?
The dining philosophers problem is useful for modeling processes competing for exclusive access to a limited number of resources, such as input-output devices, read-write operations.

Petri nets are a popular and useful model for representing discrete dynamic systems. They are named after Carl Adam Petri, who developed them in the early 1960s at the University of Bonn in Germany. A Petri net is a directed graph with two types of nodes—places and transitions. Places are drawn as circles, while transitions are represented as bars. Directed arcs (arrows) connect places to transitions and transitions to places. For each transition, the directed arcs define its input places (from a place to a transition) and output places (from a transition to a place).

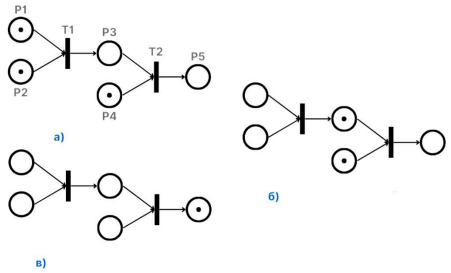
A Petri net operates by defining a marking, and then firing transitions. A marking is the distribution of tokens across the places of the Petri net. A token is represented as a small solid dot in a place on the Petri net diagram. A transition is enabled when all of its input places contain one or more tokens. A transition fires by removing one token from each input place and adding one token to each output place.

Petri nets can be used to model systems such as command issuance and reception, or the instruction queue of a computer processor. A transition may only be connected to a place, and a place may only be connected to a transition. A transition fires only when all its connected places have at least one token. When a transition fires, it generates one token for each of its output places.

A place in a Petri net can hold multiple tokens. A place may have several outputs, which will fire only if there are enough tokens for all outputs—i.e., one token per output. If two arcs lead from a place to a transition, then the place must contain two tokens for the transition to fire. When the transition fires, the tokens move to the corresponding output places.
If a transition has two arcs going to the same place (i.e., multiple outputs to one place), this means that when the transition fires, it will add two tokens to that place.
If a transition has two arcs going to different places, it means that when the transition fires, it will add one token to each of those places.

The complexity of a cybernetic system is determined by the number of states the system can assume and the number of parameters (connections) that influence state changes.

Example of a simple Petri net.

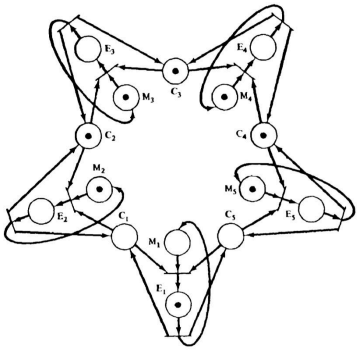


The Dining Philosophers Problem was proposed by Edsger Dijkstra. It involves five philosophers who take turns thinking and eating. The philosophers sit around a large circular table with various Chinese dishes. There is one chopstick between each pair of neighbors. However, Chinese food requires two chopsticks, so each philosopher must take the left chopstick and the right chopstick to eat.

The problem, of course, arises when all philosophers pick up the chopstick on their left and then wait for the chopstick on their right to become available—they will wait forever and starve.

The diagram shows a solution to this problem using a Petri net. The positions C1, ..., C5 represent the chopsticks, and since each one is initially available, the initial marking has a token (black dot) in each of these positions.

Each philosopher is represented by two places: Mi and Ei, corresponding to the states of thinking and eating, respectively.
For a philosopher to transition from thinking to eating, both chopsticks (left and right) must be available.

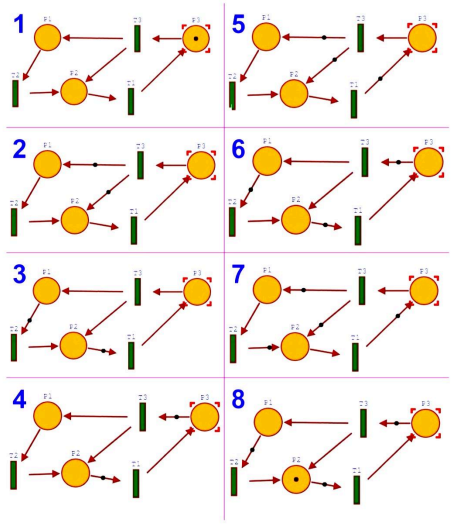


A Petri net is one of several mathematical modeling languages used to describe distributed systems. It belongs to the class of discrete event dynamic systems. A Petri net is a directed bipartite graph consisting of two types of elements: places and transitions, represented as white circles and rectangles, respectively.

A place can contain any number of tokens, which are shown as black dots. A transition is enabled if all of its input places contain at least one token. A Petri net consists of places, transitions, and arcs connecting them. Arcs go from a place to a transition or from a transition to a place, but never between places or between transitions.

Places from which arcs lead to a transition are called the input places of the transition. Places to which arcs go from a transition are called the output places of the transition.

Graphically, places in a Petri net can hold a discrete number of tokens. Any distribution of tokens among places represents the configuration of the net, called a marking. A transition in a Petri net may fire if it is enabled—that is, if there are enough tokens on all its inputs. When the transition fires, it consumes the necessary input tokens and generates tokens at its output places.



Dining Philosophers Problem and Elixir processes

Imagine five philosophers sitting around a circular table. Between each pair of philosophers is a single fork. To eat, a philosopher needs both the fork to their left and the one to their right. But

philosophers spend their time either thinking or eating—and they must not eat at the same time as their neighbors, because they share forks.

This problem is a classic example used to teach concurrency—how multiple entities (in this case, philosophers) can work independently but also share some resources (forks). The challenge is to do this without running into problems like deadlocks, where everyone is waiting forever, or starvation, where someone never gets to eat.

Now let’s connect this to Elixir.

In Elixir, we build concurrent programs using processes—but not operating system processes. Elixir (built on the BEAM virtual machine) uses lightweight, isolated processes that communicate through message passing. They don’t share memory and are very cheap to create and destroy.

So we can think of each philosopher as an Elixir process. Each one can be written as a `GenServer` or a `Task`. The philosopher process has its own state—it might be thinking, waiting, or eating.

Next, each fork can also be its own process. When a philosopher wants to eat, they send a message to both forks (the fork to their left and the one to their right), asking if they can use them. If both forks respond "yes", the philosopher eats. When done, the philosopher sends messages back to the forks saying "I'm done", and goes back to thinking.

This way, the philosophers (Elixir processes) coordinate access to shared forks (also Elixir processes). No shared memory is involved; everything is done via messages. This models real-world Elixir systems well—different parts of your system working independently, but coordinating by sending messages.

If you make all philosophers ask for their left fork first and then their right fork, you can get deadlock—all are waiting forever. This teaches you about the importance of designing safe protocols when sharing resources in concurrent systems.

Elixir makes this kind of modeling very natural. It even encourages you to think like this when designing software—use many small processes that do one thing, communicate clearly, and crash safely (thanks to supervisors).

To solve the Dining Philosophers Problem in Elixir, we need to manage how philosophers access shared forks to avoid deadlock and starvation. Here's an understandable explanation of how to do that, step by step, using Elixir processes.

Goal
Let each philosopher eat without causing:

- Deadlock (everyone waiting forever)
- Starvation (someone never eats)

Solution Idea
Instead of having all philosophers grab both forks at the same time, we design a safe protocol:

1. Each philosopher is a process (e.g. a `GenServer` or `Task`).
2. Each fork is a process that can be held by only one philosopher at a time.
3. Philosophers ask for permission to take forks.
4. Forks respond with availability.
5. Philosophers only eat if both forks say “OK”.
6. Philosophers release the forks after eating.

Avoiding Deadlock
We can break the deadlock loop by changing how forks are picked up:
- 4 philosophers ask for the left fork first, then the right.
- But 1 philosopher does the reverse: asks for the right fork first, then the left.

This small change avoids the circular waiting condition that causes deadlock.

Code Strategy (Conceptual)
Here’s what this looks like in Elixir terms:

Fork process
A `GenServer` that handles two messages: `:take` and `:put_back`.

```elixir
defmodule Fork do
  use GenServer

  def start_link(), do: GenServer.start_link(__MODULE__, :available)

  def handle_call(:take, {from, _}, :available), do: {:reply, :ok, {:taken, from}}
  def handle_call(:take, _from, {:taken, _owner}), do: {:reply, :busy, {:taken, _owner}}
  def handle_cast(:put_back, _), do: {:noreply, :available}
end
```

Philosopher process
Each philosopher tries to take both forks, then eats, then puts forks back.

```elixir
defmodule Philosopher do
```

```
  def start(name, left_fork, right_fork, reverse_order \\ false) do
    Task.start(fn -> loop(name, left_fork, right_fork, reverse_order) end)
  end

  defp loop(name, left, right, reverse) do
    think(name)
    if reverse do
      try_eat(name, right, left)
    else
      try_eat(name, left, right)
    end
    loop(name, left, right, reverse)
  end

  defp think(name) do
    IO.puts("#{name} is thinking")
    Process.sleep(:rand.uniform(1000))
  end

  defp try_eat(name, first, second) do
    case GenServer.call(first, :take) do
      :ok ->
        case GenServer.call(second, :take) do
          :ok ->
            IO.puts("#{name} is eating")
            Process.sleep(:rand.uniform(1000))
            GenServer.cast(first, :put_back)
            GenServer.cast(second, :put_back)
          :busy ->
            GenServer.cast(first, :put_back)
            IO.puts("#{name} couldn't eat (second fork busy)")
        end
      :busy ->
        IO.puts("#{name} couldn't eat (first fork busy)")
    end
  end
end
```

Setup
You create 5 forks and 5 philosophers, assigning each philosopher two forks. One philosopher reverses the order.

```elixir
{:ok, f1} = Fork.start_link()
{:ok, f2} = Fork.start_link()
{:ok, f3} = Fork.start_link()
{:ok, f4} = Fork.start_link()
{:ok, f5} = Fork.start_link()

Philosopher.start("P1", f1, f2)
Philosopher.start("P2", f2, f3)
Philosopher.start("P3", f3, f4)
Philosopher.start("P4", f4, f5)
Philosopher.start("P5", f5, f1, true)  # Reverse fork order to prevent deadlock
```

Summary
By modeling both philosophers and forks as processes and using message passing, you create a safe and concurrent system. The key ideas:
- Use Elixir processes to represent independent entities.
- Use a communication protocol to avoid deadlock.
- Control fork acquisition order for safety.

CI/CD Setup – Theory

CI/CD is a set of practices and tools that automate the processes of software integration and delivery. CI (Continuous Integration) and CD (Continuous Delivery or Continuous Deployment) enable fast and safe deployment of applications, increasing efficiency and reducing the number of errors.

The key question: How long does it take in your project to deploy a verified piece of code to the production server?

You create a Git repository on GitLab, GitHub, or similar. You have a main branch called `master` (or `main`). From the main branch, you create a `develop` branch. Then you create a `feature/...` branch from `develop` to write the new functionality. As a developer, you ensure everything works well in the branch and create a merge request to merge `feature` into `develop`. Your colleagues review your work during **code review**. Your team follows shared coding rules and style guides.

Ideally, tasks should be broken down so they take no more than 16 hours before independent testing by another team member.

You want a tester to verify your work before you move the task to "done" status. But when exactly should the tester do this — in the `feature` branch before merging, or in `develop` after merging?

Testing in `feature` branches has its challenges. For example: Where do you test? Should every branch be deployed? Will the tester switch between branches locally? Also, such `feature` branches are not yet integrated into the main code in `develop`, making long-lived branches harder to merge due to lack of synchronization.

To standardize the testing branch, we test all tasks in one branch — typically `develop`, after merging the `feature` branches. This simplifies CI/CD and ensures code is integrated.

Some teams use a different approach: all `feature` branches are merged into `develop`, but testers test a separate `test` branch. The `test` branch is created from `develop` and updated daily or on demand. This supports temporary code freeze or a specific environment. However, this complicates CI/CD, reduces flexibility, and makes changes harder. Fixes require merging both `develop` and `test`.

Thus, we test the integrated `develop` branch after merging changes. There's also GitFlow, with `release` and `production` branches. But we'll focus on the trunk-based model. Improper use of GitFlow can break CI/CD and lead to integration hell.

Automated testing helps avoid repeated manual regression testing.

A unit is an isolated component or module of a system. A unit test is designed to test a single unit in isolation, aiming to cover it as thoroughly as possible. All defects discovered by a unit test should originate from within the unit under test.

If a unit is composed of other already-tested units, writing additional unit tests for it may be redundant. In such cases, it's more appropriate to write integration tests that verify how these tested units interact within the larger composite unit. Integration tests assume that the individual units have already been tested in isolation.

While some additional unit testing might still be necessary in complex cases, most of the coverage for such higher-level components should come from integration tests.

Since defects are best identified as early as possible, it's preferable to catch them at the lowest possible level. Unit tests, being the lowest level in the testing hierarchy, serve this purpose. That's why they form the foundation of the testing pyramid.

In the **trunk-based model** for Git, the idea is simple. There's one main branch — `main` (the trunk). It’s used for development, integration, and testing. When ready to release, you create a new branch from it, test it thoroughly, and deploy it to production. Hotfixes and bug fixes are done only on the main branch (trunk) and through it. If a bug is found in production, the fix is made in `main`, and a new production branch is created or an existing one is updated. For upcoming changes, we use **feature flags** — environment variables that determine which parts of the code are active. There can be hundreds of flags, but with careful release planning, this number can be minimized.

In the trunk-based model, **automated unit testing and other test types are essential**, rather mandatory than optional. Code should not be accepted without test coverage. This should be part of the **Definition of Done (DoD).**

Merging is done through code review after the pipeline with automated tests and a linter passes. There should also be **smoke testing** of the build. Changes must go through a build process and never be deployed directly.

The team selects a coding style guide, e.g., **Airbnb JavaScript Style Guide**, to maintain a consistent code format.

Then, a **Docker image** is created and deployed to the server. The server must implement **Blue-Green Deployment**.

For databases, we use a dedicated service on known cloud platforms. During the pipeline, a containerized database can be launched for test runs.

Each time a developer commits to the repository, the CI/CD system triggers automatic deployment of a new test environment — a container or virtual machine created with tools like Docker or Kubernetes.

This **test environment is isolated** from other environments (e.g., staging or production), allowing changes to be verified without affecting other parts of the system.

We keep **backend and frontend in separate repositories**, combining them into a **monorepo only for micro-frontends** or backends with multiple **microservices**.

On the server, we configure a **cluster with autoscaling** and **Blue-Green Deployment**.

Typically, a **cloud-based cluster** is created. A service is deployed in the cluster based on our task, referencing the Docker image stored in an online image registry.

A cluster may have multiple services, each with several containers. This supports traffic routing via **load balancer** and **Blue-Green Deployment**.

We always collect **server and application logs**.

It's important that **development, staging, and production environments** are deployed from the same **codebase**, possibly using different branches.

If there are multiple codebases, it's not just an app but a **distributed system**. Each component is a separate program and may follow **Twelve-Factor App** principles.

All **static files** should be served via a **CDN**.

The application must run as **one or more stateless processes**. Twelve-factor processes don't share state. All persistent data should be stored in external services, typically a database. We avoid sessions and similar mechanisms.

In **Node.js**, a **cluster** is a way to organize multiple processes within a container or EC2 instance. Since Node.js runs in a single thread, the `cluster` module allows for creating multiple workers that handle requests in parallel, distributing load across CPU cores. An **AWS cluster** in this context doesn't directly relate to Node.js clustering but rather helps **scale** the number of containers or instances running the Node.js app. Internal processes use the `cluster` module.

BEAM (Erlang and Elixir VM) provides **built-in process clustering**. In BEAM, each process is lightweight, independent, and parallel. BEAM nodes (execution instances) can form a cluster. With Elixir and BEAM, you can directly create a process cluster that automatically communicates with other nodes. AWS clusters (ECS or EC2) are used to run multiple BEAM nodes, while BEAM manages process coordination itself — unlike Node.js, which requires manual clustering.

To avoid **Single Point of Failure (SPoF)** in information systems, architecture should be designed so that the failure of at least two independent components is required to bring the system down.

Kubernetes

Kubernetes is an open-source platform for automating the deployment, scaling, and management of containerized applications. It allows developers to manage containers at scale, ensuring they run efficiently, reliably, and with minimal downtime. Kubernetes abstracts away the complexity of managing infrastructure and enables the orchestration of containers across multiple servers.

At its core, Kubernetes uses a set of objects like Pods, Services, and Deployments to define the desired state of an application. A Pod is the smallest deployable unit in Kubernetes, typically running a single container. Services provide a stable endpoint to access one or more Pods,

while Deployments manage the lifecycle of Pods, ensuring that the desired number of replicas are running at all times.

Kubernetes helps to build fault-tolerant systems through several key features. One of the most crucial is its self-healing capability. If a Pod fails or crashes, Kubernetes automatically reschedules it onto another node in the cluster without any manual intervention. This ensures that the application remains available and resilient to failures.

The system also supports horizontal scaling, which means that when the load on an application increases, Kubernetes can automatically spin up new instances of Pods to handle the increased traffic. Similarly, when traffic decreases, Kubernetes can scale down the number of Pods to save resources. This scalability is essential for maintaining performance and reliability during varying loads.

Kubernetes leverages replication and distribution of containers across multiple nodes, reducing the risk of failure. By running multiple copies of a Pod across different nodes, it ensures that if one node goes down, there are still healthy replicas available to take over the workload. This redundancy contributes to the system's high availability.

In addition to its self-healing and scaling features, Kubernetes provides built-in load balancing. The Service object can distribute incoming traffic across the Pods it manages, ensuring that no single Pod is overwhelmed. This load balancing ensures consistent performance and helps mitigate failures caused by overloaded Pods.

Kubernetes also supports declarative configuration, which means that users define the desired state of the application, and Kubernetes works to ensure that the actual state matches. This configuration can include resource limits, replica counts, and network policies, making it easier to maintain consistent behavior across environments.

Furthermore, Kubernetes integrates with monitoring and logging tools, allowing teams to monitor the health of applications and the infrastructure that supports them. These insights are critical for identifying potential issues before they affect users and enable proactive management of the system.

Finally, Kubernetes offers flexibility in terms of fault tolerance by supporting multi-cloud and hybrid cloud deployments. This enables organizations to run applications across multiple cloud providers or on-premise data centers, ensuring resilience even if one provider experiences an outage. This versatility is a significant advantage for building fault-tolerant, distributed systems.

The main concepts of Kubernetes are:

1. Pod: The smallest deployable unit in Kubernetes, typically running a single container. Pods can host one or more containers that share resources like networking and storage.

2. Node: A physical or virtual machine that runs one or more Pods. Each node contains the services necessary to run Pods, such as the container runtime and the Kubernetes agent (kubelet).

3. Cluster: A set of nodes grouped together to run applications. Kubernetes manages clusters by scheduling Pods across the available nodes.

4. Deployment: Manages the desired state of Pods, ensuring that the specified number of replicas of a Pod are running at all times. It handles updates, rollbacks, and scaling of Pods.

5. Service: A stable endpoint that exposes a set of Pods to external traffic or other services inside the cluster. Services provide load balancing and abstract the complexity of accessing individual Pods.

6. ReplicaSet: Ensures a specified number of identical Pods are running at any given time. It is typically used by Deployments to maintain the desired state.

7. Namespace: A way to partition the Kubernetes cluster into multiple virtual clusters, which can be useful for isolating resources and managing large-scale environments.

8. ConfigMap and Secret: Used to manage configuration data for applications. ConfigMaps store non-sensitive data, while Secrets store sensitive information like passwords or API keys.

9. Ingress: Manages external access to services in a cluster, typically HTTP. It allows routing traffic to different services based on the URL or host.

These concepts work together to enable efficient orchestration, scaling, and management of containerized applications in a Kubernetes environment.

Kubernetes typically runs on clusters of physical or virtual machines, either on-premises or in the cloud. The cluster consists of a control plane, which manages the overall system, and worker nodes, which handle the actual application workloads. The control plane components, like the API server, scheduler, and etcd, are critical for managing the state and scheduling of applications. Managed services like Amazon EKS, Google GKE, and Azure AKS handle the control plane for you, ensuring high availability and resilience.

For fault tolerance, Kubernetes is designed to handle node failures by automatically rescheduling pods to healthy nodes. It relies on etcd for state persistence, which is typically replicated across multiple nodes to prevent data loss. Control plane components can also be run in high-availability configurations, distributing them across multiple nodes to avoid single points of failure. In self-managed clusters, operators often use load balancers and multiple availability zones to further reduce the risk of outages.

However, if the entire control plane fails, recovery depends on the underlying infrastructure and backup strategies. Managed services automatically handle this, while self-managed clusters require regular etcd backups and disaster recovery plans, including automated cluster creation scripts. This approach ensures that even in the worst-case scenario, the cluster can be quickly restored to a functional state.

Docker and Kubernetes are both essential tools for modern containerized application development, but they serve different purposes:

Docker:
Docker is a platform used to create, deploy, and run applications in containers. Containers package up the application and all its dependencies, ensuring that the application runs consistently across different environments.

Key Features of Docker:
- Containerization: Allows you to package applications and their dependencies into a single, portable container image.
- Isolation: Containers are isolated from the host system and other containers, which ensures security and consistent performance.
- Portability: Since Docker containers include everything an app needs, they can run anywhere — from your local machine to production servers, without modification.
- Efficiency: Docker containers share the host OS kernel, making them more lightweight and faster to start than traditional virtual machines.

Kubernetes:
Kubernetes (often abbreviated as K8s) is an open-source platform for managing containerized applications in a clustered environment. It automates the deployment, scaling, and operation of application containers.

Key Features of Kubernetes:
- Orchestration: It helps in managing and orchestrating a large number of containers across multiple servers, ensuring they work together smoothly.
- Scaling: Kubernetes can automatically scale your application up or down depending on load.
- Self-healing: It can automatically replace failed containers and reschedule them on healthy nodes.
- Load balancing: Distributes traffic to your containers effectively, ensuring high availability.

How Docker and Kubernetes Work Together:
- Docker is used to build and run individual containers. It's great for creating and packaging applications.

- Kubernetes is used to deploy, manage, and orchestrate those containers in a larger, distributed environment. It ensures that containers are running in the right place and scales them as needed.

To summarize:
- Docker is about packaging your app in a container.
- Kubernetes is about running and managing those containers at scale.

Docker is a powerful tool used to create, deploy, and run applications by packaging them into containers. Containers are lightweight, portable, and can run consistently across various environments. Below is a deeper look into Docker, its components, and its usage.

Core Concepts of Docker:

1. Containers:
   Containers are isolated environments where applications run. They are a form of virtualization that is more lightweight than traditional virtual machines (VMs). Unlike VMs, containers share the host OS kernel, making them faster to start and more efficient in terms of resource usage.

2. Docker Images:
   A Docker image is a lightweight, stand-alone, and executable package that includes everything needed to run a piece of software (code, runtime, libraries, environment variables, and config files).
   - Docker images are the blueprints used to create containers.
   - Images are read-only. When you run an image, Docker creates a container, which is a writable layer on top of the image.

3. Dockerfile:
   A Dockerfile is a script containing instructions to build a Docker image. It defines the environment the application will run in and the steps required to build the image. For example, you can specify which base image to use, copy files into the image, install dependencies, and set environment variables.

   Basic structure of a Dockerfile:
   ```dockerfile
   # Use an official Node.js runtime as a parent image
   FROM node:14

   # Set the working directory in the container
   WORKDIR /app

   # Copy the current directory contents into the container
   ```

```
COPY . .

# Install dependencies
RUN npm install

# Make port 3000 available
EXPOSE 3000

# Define the command to run the app
CMD ["npm", "start"]
```

4. Docker Hub:
   Docker Hub is a cloud-based registry where Docker images are stored and shared. It's the default registry used by Docker to pull images. You can search for and pull pre-built images from Docker Hub, or you can push your own images to share with others.

5. Docker CLI:
   Docker provides a command-line interface (CLI) to interact with Docker. Some commonly used Docker commands include:
   - `docker build`: Builds a Docker image from a Dockerfile.
   - `docker run`: Runs a container from a specified image.
   - `docker ps`: Lists running containers.
   - `docker stop`: Stops a running container.
   - `docker images`: Lists Docker images available locally.

6. Docker Compose:
   Docker Compose is a tool used to define and run multi-container Docker applications. Using a `docker-compose.yml` file, you can specify the services that make up your app, including the build context, environment variables, network settings, and more.

   Example `docker-compose.yml`:
   ```yaml
   version: '3'
   services:
     web:
       build: .
       ports:
         - "5000:5000"
     db:
       image: postgres
       environment:
         POSTGRES_PASSWORD: example
   ```

7. Volumes:
Docker volumes are used to persist data outside of the container's lifecycle. By default, data inside containers is lost when they are removed. Volumes ensure that important data is retained even when containers are removed or recreated. Volumes can be mounted into containers to provide persistent storage.

8. Networking:
Docker containers communicate with each other through a virtual network. By default, Docker creates a bridge network that allows containers to communicate with each other using their IP addresses. You can create custom networks for more complex applications and control the communication between different containers.

9. Docker Registries:
A Docker registry is a place where Docker images are stored and distributed. Docker Hub is the default registry, but you can also use private registries or self-hosted ones like Harbor.

How Docker Works:

1. Build:
- You write a Dockerfile to specify the image configuration.
- You run `docker build` to create the image.

2. Run:
- You run the image to create a container using the `docker run` command.
- You can specify various options, such as exposing ports or mounting volumes.

3. Manage:
- Once running, you can interact with your container, check its logs (`docker logs`), or even execute commands inside the container (`docker exec`).

4. Share:
- If you want to share your image, you can push it to Docker Hub or a private registry using the `docker push` command.

Benefits of Docker:

1. Portability: Docker containers can run on any machine that has Docker installed, regardless of the underlying OS or hardware, as long as Docker is supported.

2. Consistency: Since containers encapsulate all dependencies, the application behaves the same way on all environments (development, staging, production).

3. Efficiency: Containers are lightweight compared to VMs because they share the host OS kernel.

4. Scalability: Docker works well in distributed systems. When combined with orchestration tools like Kubernetes, it provides powerful ways to scale applications automatically.

5. Isolation: Docker ensures that each container runs in isolation, reducing the risk of conflicts between dependencies or libraries.

Use Cases of Docker:

- Microservices: Docker is often used for building microservices-based applications where each microservice runs in its own container.

- CI/CD Pipelines: Docker helps in automating the software delivery process by providing a consistent environment for building, testing, and deploying applications.

- Development and Testing: Developers can use Docker to replicate production environments locally, ensuring that the code behaves the same way when deployed.

- Environment Replication: You can replicate environments across different machines, ensuring that everyone works with the same setup.

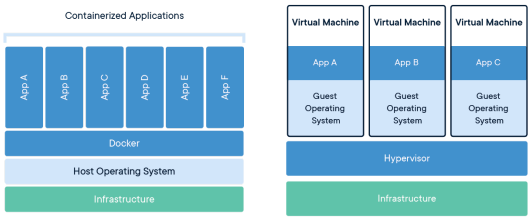
Docker vs Virtual Machines:

While both Docker and VMs allow you to isolate applications, Docker is more efficient. Unlike virtual machines, which run full operating systems, Docker containers share the host system's kernel and only package the application and its dependencies, making them lightweight and fast to deploy.

Containers are an abstraction at the app layer that packages code and dependencies together. Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space. Containers take up less space than VMs (container images are typically tens of MBs in size), can handle more applications and require fewer VMs and Operating systems.

Virtual machines (VMs) are an abstraction of physical hardware turning one server into many servers. The hypervisor allows multiple VMs to run on a single machine. Each VM includes a full copy of an operating system, the application, necessary binaries and libraries – taking up tens of GBs. VMs can also be slow to boot.

Containers and VMs used together provide a great deal of flexibility in deploying and managing app.



Team

A team is not merely a collection of individuals; it is an indivisible entity with emergent properties that arise from collaboration and shared purpose. Its value lies not in the sum of its parts but in the synergy created when members work together effectively. Collective knowledge surpasses individual knowledge only when every voice is heard and considered. When some perspectives are ignored, critical insights may be lost, and the full potential of the team is diminished. Importantly, collective knowledge is not the same as common knowledge or a watered-down compromise. It is the integration of all individual contributions into a richer, more complete understanding.

Strength lies in diversity, not in uniformity. Evolution itself favors this principle. It is through diverse perspectives, experiences, and approaches that teams discover innovative solutions, learn from one another, and achieve more than they ever could in an environment where everyone thinks the same.

A truly cross-functional team is one that possesses all the skills necessary to deliver a complete product increment—not just to complete a step in a production pipeline. This means the team is capable of planning, building, testing, and delivering value end-to-end. Estimation of tasks should be a shared activity, involving all team members regardless of their specific expertise. This ensures that the team builds a common understanding of the work and reinforces collective ownership.

Being cross-functional does not mean that each member must be a universal "full-stack" specialist. Rather, it means that the team as a whole must be capable of accomplishing the work. Specialization is fine—valuable, even—as long as there is enough overlap, communication, and collaboration to complete any task the team commits to.

When it comes to task estimation, teams should collaborate to decompose and evaluate tasks together. Instead of relying on mythical "man-hours," which falsely suggest that time and people are interchangeable, agile teams often use story points as a unit of measure. Story points reflect the relative complexity, risk, and effort of a task, not the exact duration it will take to complete.

To calibrate story points effectively, it is helpful to choose a reference task—a piece of work the entire team understands well—and assign it a baseline value, typically one story point. From there, tasks that are approximately twice as difficult or twice as large can be estimated at two story points, and so on. This comparative approach enables teams to estimate different kinds of work—development, design, testing, documentation—using a shared understanding of complexity, even when the nature of the work varies.
Having a well-understood reference task ensures that team members, even with different areas of expertise, can estimate work consistently across various domains—be it frontend, backend, testing, or design.

Ultimately, cross-functional teams thrive when they share responsibility for planning, estimating, and delivering. The practices of open communication, collective estimation, and mutual respect for each member's knowledge contribute to a resilient, adaptive, and productive team environment.

Providing fair compensation (salary according to the market standards) and promoting a healthy work-life balance are key factors in building a successful and resilient team. Avoiding excessive overtime and maintaining a sustainable pace help prevent burnout and foster long-term productivity.

Implementing modern methodologies such as Scrum, Extreme Programming (XP), and DevOps can significantly enhance team management. These frameworks offer proven approaches to progress tracking, continuous improvement, and risk mitigation—contributing to a more adaptive, efficient, and motivated development environment.

Mentoring and education

Education has a purpose or goal, such as helping students gain knowledge, develop skills, and become responsible members of society. Because of this, the quality of education is measured by how well it achieves that goal. If students are not learning effectively or achieving expected outcomes, then the education system or method may not be considered high quality.

To achieve this goal successfully, education must take into account the students’ current level of understanding and individual abilities. Not all students learn the same way or at the same pace. So, effective education adapts to students' needs and finds the best way to present the material.

A key principle is moving from simple to complex. Teachers should first introduce basic concepts that are easy to understand and then gradually move to more difficult topics. This step-by-step

approach helps students build a strong foundation and better understand advanced material later on.

Students learn best when they are actively involved—through discussion, exploration, problem-solving, and hands-on activities—not just passively listening.

When a student asks a question, it shows they are thinking, curious, and engaged. Every question is a step toward understanding, even if it seems basic or obvious to others. What might seem simple to one person could be confusing to someone else, and that's completely normal.

If teachers or classmates laugh at a question or dismiss it, the student might feel embarrassed and stop asking — and that shuts down learning. A good learning environment encourages all questions, because it helps everyone explore, clarify, and deepen their understanding.

The best teachers teach with humility, patience, and care, because their success is measured not by how much they shine — but by how much their students do.

Sources:

https://12factor.net/
https://trunkbaseddevelopment.com/
https://nvie.com/posts/a-successful-git-branching-model/
http://www.extremeprogramming.org/
https://docs.aws.amazon.com/whitepapers/latest/cicd_for_5g_networks_on_aws/cicd-on-aws.html
https://about.gitlab.com/topics/version-control/what-are-gitlab-flow-best-practices/
https://monorepo.tools/
https://kubernetes.io/

Databases

Databases are a cornerstone of modern data management, serving as structured repositories for storing, retrieving, and managing information. In the digital age, the ability to efficiently handle large volumes of data is crucial for organizations of all sizes. To understand databases fully, it is essential to grasp the fundamental concepts surrounding data, database structures, and the various technologies that support them.

DIKW Pyramid
Data is the raw, unprocessed facts and figures that form the basis of all information systems. In the context of the DIKW pyramid (Data, Information, Knowledge, Wisdom), data represents the

foundational layer, which gains context as it moves up the hierarchy. Information is data organized to provide meaning, knowledge is the understanding derived from information, and wisdom is the insightful application of knowledge.

A database is a structured collection of data, designed to efficiently store, retrieve, and manage information. It can take various forms, from simple text files like CSV (Comma-Separated Values) to complex, highly structured systems like SQL databases. CSV files are straightforward text-based formats where data is organized in rows and columns, making them easy to read and process but lacking the advanced features of a full database management system (DBMS).

SQL (Structured Query Language) is a standardized language for managing relational databases. It allows users to query, insert, update, and delete data efficiently. Popular SQL-based systems include PostgreSQL and MySQL. These databases are characterized by their use of tables with predefined schemas, ensuring data consistency and integrity through mechanisms like ACID (Atomicity, Consistency, Isolation, Durability) transactions.

ACID properties are essential for maintaining data reliability in relational databases. They ensure that transactions are processed reliably, even in the presence of errors. Meanwhile, the CAP theorem, which stands for Consistency, Availability, and Partition Tolerance, highlights the trade-offs in distributed database systems, stating that it is impossible for a distributed system to simultaneously guarantee all three properties.

In computer programming, create, read, update, and delete (CRUD) are the four basic operations (actions) of persistent storage
The term C.R.U.D was likely first popularized in 1983 by James Martin in his book Managing the data-base environment.

To reduce data redundancy and improve efficiency, relational databases often undergo normalization, a process that structures data into normal forms. This approach minimizes duplicate data and maintains data integrity. Venn diagrams are often used to illustrate the relationships between different sets of data in a database.

Sharding and Eventual Consistency
As databases grow, they may require sharding – the process of partitioning data across multiple servers to improve performance and scalability. However, distributed systems that use sharding often rely on eventual consistency, meaning data will become consistent over time, rather than guaranteeing immediate consistency like traditional relational systems.

NoSQL databases provide an alternative to relational databases, focusing on flexible, scalable, and high-performance data storage. They include document-oriented databases like MongoDB, which stores data in JSON-like documents. Key concepts here include cursors, collections, documents, and aggregation, offering powerful ways to query and analyze unstructured data.

Unlike ACID, NoSQL systems often follow the BASE (Basically Available, Soft state, Eventually consistent) model, prioritizing scalability over strict consistency.

Vector databases, graph databases, and row- and column-oriented databases represent more specialized database architectures. Vector databases are optimized for handling high-dimensional data, while graph databases like Neo4j excel at representing complex relationships. Row and column databases, such as HBase and Cassandra, are designed for high-speed reads and writes across massive datasets.

Cloud and Managed Database Solutions
Cloud platforms like Amazon Web Services (AWS) offer managed database services, including DocumentDB, DynamoDB, and Amazon Aurora. These services provide scalable, high-performance storage options without the overhead of traditional database management, allowing businesses to focus on their core applications.

Understanding the wide variety of database technologies, from traditional SQL systems to modern NoSQL and specialized database types, is essential for managing today's data-intensive applications. As data continues to grow in volume and complexity, choosing the right database architecture becomes a critical decision for any organization.

The English computer scientist Edgar Codd (1923 – 2003) described the relational database in 1969.
A relational database is based on relations and connections (relationships), or tables and connections, as defined by Edgar Codd.
The core concept of relational databases is that they are built on a structure called a "relation." A "relation" is the fundamental structuring element of relational databases. In essence, databases consist of "relations" (tables) and conditional links between them. A "relation," famously known as a "table," represents an ordered set (tuple).

The generally accepted definition of a relation in mathematics is as follows:
Given sets S1, S2,..., Sn (not necessarily distinct), R is a relation on these n sets if it is a set of n-tuples, the first component of which is drawn from S1, the second component from S2, and so on.
More concisely, R is a subset of the Cartesian product S1 × S2 x . . . × Sn. Relation R is said to be of degree n. Each of the sets S1, S2, . .., Sn on which one or more relations are defined is called a domain.

A Relational Database Management System (RDBMS) is a type of database based on the relational data model. The relational data model uses tables consisting of rows and columns to store and organize data. Each table in a relational database has a set of columns representing data attributes and rows representing specific records or tuples.

Databases that are not normalized, do not adhere to ACID principles, and are not based on rows are not considered relational. For example, there are column-based databases like Apache Cassandra.

Comma-separated values (CSV) is a text file format that uses commas to separate values, and newlines to separate records. A CSV file stores tabular data (numbers and text) in plain text, where each line of the file typically represents one data record.

CSV (Comma-Separated Values)

A simple CSV file, `users.csv`:

```
id,name,email,age
1,John Doe,john@example.com,30
2,Jane Smith,jane@example.com,25
3,Bob Brown,bob@example.com,40
```

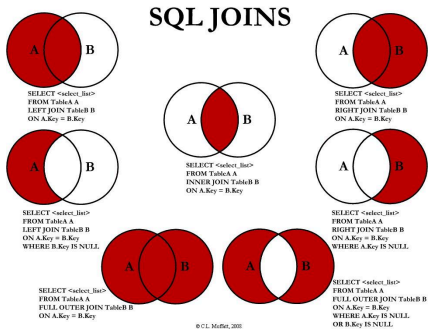
CRUD SQL (Relational Database)

For a SQL database like MySQL or PostgreSQL:

```
-- Create
CREATE TABLE users (
  id INT PRIMARY KEY,
  name VARCHAR(255),
  email VARCHAR(255),
  age INT
);

INSERT INTO users (id, name, email, age) VALUES
(1, 'John Doe', 'john@example.com', 30),
(2, 'Jane Smith', 'jane@example.com', 25),
(3, 'Bob Brown', 'bob@example.com', 40);

-- Read
SELECT * FROM users;

-- Update
UPDATE users SET age = 35 WHERE id = 1;

-- Delete
```

DELETE FROM users WHERE id = 3;

SQL joins appear to be set-based, the use of Venn diagrams to explain them seems, at first blush, to be a natural fit.



CRUD MongoDB (NoSQL Database)

Using MongoDB's `users` collection:

```
// Create
db.users.insertMany([
  { id: 1, name: "John Doe", email: "john@example.com", age: 30 },
  { id: 2, name: "Jane Smith", email: "jane@example.com", age: 25 },
  { id: 3, name: "Bob Brown", email: "bob@example.com", age: 40 }
]);

// Read
db.users.find();

// Update
db.users.updateOne(
  { id: 1 },
  { $set: { age: 35 } }
);

// Delete
db.users.deleteOne({ id: 3 });
```

An **index** in a database is a data structure that improves the speed of data retrieval operations on a table at the cost of additional storage space and maintenance. It works much like an index in a book — instead of scanning the entire table to find specific rows, the database can use the index to quickly locate the data. Indexes are commonly created on one or more columns of a table that are frequently searched, sorted, or used in join conditions. They are especially useful for large databases where full table scans would be too slow.

Internally, most databases use B-trees or hash tables to implement indexes. When a query is executed, the database checks if there's an index that matches the query conditions. If found, it navigates the index to locate the relevant data efficiently, avoiding a full scan of the table. However, indexes also add overhead during insert, update, or delete operations, because the index itself needs to be updated accordingly. Therefore, while indexes significantly boost read performance, they should be used judiciously to balance read and write efficiency.

Sources:

Codd, E. F. The Relational Model for Database Management: Version 2. Addison-Wesley, 1990.

García-Molina, Héctor, Jeffrey Ullman, and Jennifer Widom. Database Systems: The Complete Book. Pearson, 2008.

Banker, Kyle. MongoDB in Action. Manning Publications, 2016.

Groff, James R., and Paul N. Weinberg. SQL: The Complete Reference, 3rd Edition. McGraw-Hill, 2009.

UI/UX

What Is UI/UX?

UI (User Interface) is the visual and interactive part of an application—the buttons, forms, text, and layouts users see and touch.

UX (User Experience) is how users feel when interacting with the interface—how easy, fast, and enjoyable it is to achieve their goals.

Together, UI/UX design aims to make digital products efficient, accessible, and pleasant to use.

Core Principles of UI/UX Design

The main principle behind UI/UX design is **convenience**. A good user interface should be:

- Operable – easy to navigate and interact with.
- Controllable – users should always feel in control of their actions.
- Understandable – the interface should be intuitive and self-explanatory.
- Readable – content should be clear, legible, and well-structured.
- Consistent – behavior and visuals should remain uniform across screens.
- Informative – the UI should clearly communicate status, feedback, and next steps (e.g., loading indicators, error messages, tooltips).
- Efficient — delivering high-quality outcomes with speed and precision

One-Time Effort, Repeated Impact
A developer might spend 5 hours optimizing a page load or simplifying a workflow. But if that app is used by 10,000 users a day, and each user saves just 10 seconds, that's:
10,000 users × 10 seconds = ~28 hours saved every day.
That's a huge return on the developer's one-time effort.

Convenience is a main principle.

Accessibility

Accessibility in UI/UX means designing digital products — like websites and apps — so that everyone can use them, including people with disabilities.

While most user interfaces are built for the "average" user, not everyone sees, hears, moves, or processes information the same way. Some users may have visual impairments, hearing loss, motor difficulties, or cognitive challenges.

Designing with accessibility in mind ensures that people with these disabilities can still navigate, understand, and interact with the interface independently and effectively.

A design system in UI/UX is a collection of reusable components, patterns, and guidelines that help teams create consistent, efficient, and user-friendly interfaces. By standardizing elements like buttons, typography, colors, and layout structures, a design system ensures a cohesive visual and functional experience across all parts of a product. This not only improves the user experience by providing familiarity and predictability but also reduces the time spent making design decisions from scratch. Designers and developers can work faster by pulling from a shared library, leading to more efficient collaboration and quicker iterations.

Beyond speed and consistency, design systems also support scalability and maintainability. As teams and products grow, a design system keeps everyone aligned, reduces UI inconsistencies,

and helps enforce best practices, including accessibility. Updates can be made globally, ensuring changes are applied across all components without redundant work. In short, design systems are essential for building high-quality digital products that are scalable, accessible, and efficient to develop and maintain.

Modern UI/UX Development: Key Characteristics

Today's frontend best practices include:

- Component-based architecture – UI is broken into self-contained, reusable pieces.
- Utility-first or CSS-less styling – Using frameworks like TailwindCSS instead of writing large custom CSS files.
- Immutable UI logic – Interfaces are predictable and based on state, often managed with unidirectional data flow.
- Single source of truth – State and styling are centralized for consistency.
- Declarative design – The UI reflects the current state automatically, reducing bugs and improving clarity.

Imagine building a user interface as if you're constructing a tree, starting from the smallest pieces and growing upward. At the very base, you have basic components — these are your fundamental building blocks, like buttons, input fields, or simple labels. Each of these components does one thing well and knows just enough to display itself properly. They don't hold big ideas or complicated logic; instead, they listen carefully to the information given to them and react accordingly.

Now, when you combine these basic components, you start to form complex components, which are essentially groups of smaller components working together. Think of a sidebar that contains buttons, icons, and links — that sidebar is itself a component, made up of many smaller parts. Layer after layer, these components nest inside one another, creating what we call a components tree. This tree represents the entire user interface structure, where each branch is a component managing its own little piece of the UI puzzle.

What makes this tree powerful is that it's self-sufficient. Each component controls its own state or relies on the state passed down from above, rather than depending on external tricks or outside technology to manipulate what's shown. This means the UI you see is always a direct

reflection of the current state of its components — nothing hidden, no magic DOM hacks, just straightforward, predictable rendering.

To keep everything organized, we follow a top-down approach. The highest-level components hold the key state — the data and logic that matters most — and they pass down information to their children through properties. The children then render themselves based on this data, sometimes managing their own small internal state but always within the context of what they receive from above. This flow of data from parent to child keeps the interface consistent and easy to reason about.

In essence, your UI is like a living tree: starting from simple roots, growing into a rich structure of components, each managing its own state, and all working together in harmony, with the entire view controlled by the state within the components, not by anything external or unpredictable.

When you use traditional CSS that styles elements globally — like targeting HTML tags or classes across the entire page — it can “crash” the neatness of your component tree. Why? Because CSS lives outside the component system. It doesn’t care about the boundaries or the internal state of your components. Instead, it applies styles blindly across the DOM, no matter where or how those elements are rendered.

This means your components lose some of their self-sufficiency. Suddenly, styles can leak in or out, causing unexpected visual changes or conflicts. A button inside one component might look different because of a global CSS rule that was meant for something else, breaking the idea that each component fully controls its own appearance based on its state.

That’s why modern component-based architectures often use scoped styles, CSS-in-JS, or style encapsulation techniques. These keep the styles contained inside the component’s “branch” of the tree, preserving the predictability and isolation of each part of the UI. When styles live and breathe inside the component, just like its state and logic, your whole tree stays stable, easy to maintain, and truly self-sufficient.

From CSS → TailwindCSS → No CSS

We began with **CSS**, the long-standing default. Developers wrote class names like `.form`, `.input`, `.btn`, and placed their styles in global or modular CSS files. This worked for a few years—clean separation of structure and presentation, reusable class definitions, and full access to pseudo-classes and complex selectors. However, some problems arose, which led to the development of approaches like BEM, Atomic CSS, SMACSS, and others. In essence, though, these were more about avoiding CSS logic than improving it.

So, CSS had a cost.

CSS required **naming things**, managing **global scope**, and often led to **leaky styles** or **specificity battles**. Sharing components across projects meant shipping styles with them, worrying about conflicts, and hoping everything stayed in sync.

But as projects grew, so did the CSS files. We duplicated similar styles, wrestled with naming conventions, and dealt with the infamous "specificity wars." Refactoring became a headache, and teams struggled to maintain consistency across components.

Then came TailwindCSS, a new philosophy: instead of abstracting styles into reusable class names, why not apply them directly with small, single-purpose utility classes?

Tailwind offered speed—you could style directly in the markup, without writing a single line of CSS. It offered consistency—spacing, colors, and fonts followed a design system by default. And it reduced bloat—thanks to its just-in-time engine, only the styles you actually used made it to production. Developers embraced it because it removed decisions around naming and helped them build faster with fewer bugs.

Enter **TailwindCSS**, a major shift in thinking.

Tailwind asked: What if you skipped writing CSS entirely? Instead of `.btn`, you use `bg-indigo-600 text-white py-2 px-4`. Instead of declaring styles globally, you compose them directly in the HTML or JSX. It's **utility-first**, **design-system-enforced**, and **JIT-powered**, generating only the styles you actually use.

This approach solved many headaches:

- No more naming conflicts
- Styles lived close to markup
- Consistent design tokens (spacing, colors, etc.)
- Fast prototyping with production-ready styling

Tailwind became a favorite for teams that needed to move fast while staying consistent.

But in some cases, even Tailwind felt like too much.

That's where **NoCSS** enters—an even more opinionated direction.

With NoCSS, styling is **fully encapsulated inside the component**. There are **no CSS files**, no class names, and no external dependencies. Styles are defined using **JavaScript objects** (like

in React's `style={{ ... }}`) or attached through frameworks (e.g., styled-components, inline style props in React Native, etc.).

Everything—from layout to typography to color—is controlled by the component itself. There are **no pseudo-classes** like `:hover` or `:focus`, unless you emulate them through component logic (e.g., via `onMouseEnter`). The component is **self-contained and portable**. It renders exactly how it's meant to look, anywhere it's used.

This style works well for:

- Design systems that demand full encapsulation
- Cross-platform codebases (React Native, Flutter)
- Apps where style logic is tightly bound to state/behavior
- Environments without CSS support (e.g., email renderers, custom UI engines)

🎯 The Core Shift

- **CSS**: You declare styles *somewhere else* and reference them.
- **TailwindCSS**: You compose styles *where you use them*.
- **NoCSS**: You *own* the entire view inside the component — logic, state, and style.

TailwindCSS: The Utility-First Philosopher

TailwindCSS sees the web not as a split between HTML and CSS, but as a unified canvas where structure and style belong together. It doesn't believe in hiding styles behind layers of abstraction or deep CSS hierarchies. Instead, it says: declare exactly what you want, where you need it.

To Tailwind, every piece of your interface should be composed of small, predictable building blocks — `text-sm`, `bg-blue-600`, `rounded-md`. These are tokens, not magic — each one representing a decision someone already made in your design system. You aren't inventing styles; you're assembling a consistent interface using a shared visual language.

You don’t name buttons like `.btn-primary`; you build them: `px-4 py-2 bg-blue-600 text-white rounded hover:bg-blue-700`. It’s not minimalism for the sake of simplicity — it’s a belief in clarity, constraint, and composability.

Tailwind trusts that if you keep style decisions close to the markup, you’ll move faster, avoid naming fatigue, and prevent the slow creep of inconsistent designs. Its philosophy: eliminate guesswork, embrace repetition, and let the system enforce your visual language.

Flutter (NoCSS): The Self-Sufficient Component

Flutter, and NoCSS approaches like it, believe in something deeper: a component should control everything about itself — structure, behavior, and style — from within.

Here, there's no CSS at all. No class names. No selectors. No inheritance chains to trace. If a button is blue, it’s because the code says it’s blue — right inside the component, right next to the logic that decides what happens when it’s clicked.

The philosophy here is: a component is a complete unit of interface. It shouldn't rely on global styles, cascading rules, or external CSS files. It should carry everything it needs to look and act the way it does — like a living, breathing piece of the UI.

This approach treats UI not as decorated HTML, but as structured data and behavior. It’s declarative, reactive, and fully in the hands of the developer. You want padding? Set `EdgeInsets.symmetric`. You want a hover effect? Write a gesture detector with a state change. Every style is a choice driven by logic, not just aesthetics.

Flutter’s philosophy is about control and encapsulation. Components aren’t just visual — they’re interactive, styled, and self-contained. It’s ideal for large, stateful applications, cross-platform systems, or environments where global CSS simply doesn’t exist (like mobile apps).

Summary of Philosophies (Narrative Contrast)

TailwindCSS believes in building consistent interfaces quickly using design system constraints and utility classes, keeping styling decisions visible and composable in your markup.

Flutter (NoCSS) believes in self-contained components where styling is part of the code — no external styles, no cascading, just logical, deliberate, full-control UI building.

Both philosophies aim to solve the same problem — scalable, maintainable UI development — but they take different roads:

Tailwind unifies structure and style in the markup.

Flutter unifies style and behavior inside the component.

Each is elegant in its own way — it just depends on whether you're building for the browser or for a world beyond CSS.

Flutter example with Dart language

```dart
import 'package:flutter/material.dart';

void main() {
  runApp(HoverExampleApp());
}

class HoverExampleApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Hover Example',
      home: Scaffold(
        appBar: AppBar(title: Text('Hover Example')),
        body: Center(
          child: HoverContainer(),
        ),
      ),
    );
  }
}

class HoverContainer extends StatefulWidget {
  @override
  _HoverContainerState createState() => _HoverContainerState();
}

class _HoverContainerState extends State<HoverContainer> {
  bool _isHovered = false;

  @override
  Widget build(BuildContext context) {
    return MouseRegion(
      onEnter: (_) => _onHover(true),
      onExit: (_) => _onHover(false),
      child: AnimatedContainer(
        duration: Duration(milliseconds: 200),
        width: 200,
        height: 100,
        decoration: BoxDecoration(
          color: _isHovered ? Colors.blueAccent :
Colors.grey[300],
          borderRadius: BorderRadius.circular(12),
        ),
        alignment: Alignment.center,
        child: Text(
          _isHovered ? 'Hovering!' : 'Hover over me',
          style: TextStyle(fontSize: 20),
        ),
      ),
    );
  }

  void _onHover(bool hovering) {
    setState(() {
      _isHovered = hovering;
    });
  }
}
```