



# Optimisation de performance bénéfice ou sacrifice ?

Emmanuel-Lin TOULEMONDE



24.01.2025 | SnowCamp



# SNOWCAMP



## Qui suis-je ?

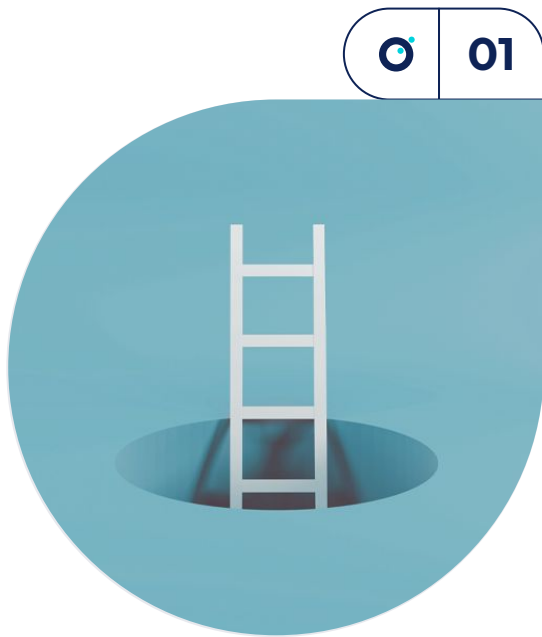


Emmanuel-Lin  
TOULEMONDE

~10 ans de Data Science, Data Eng, MLOps



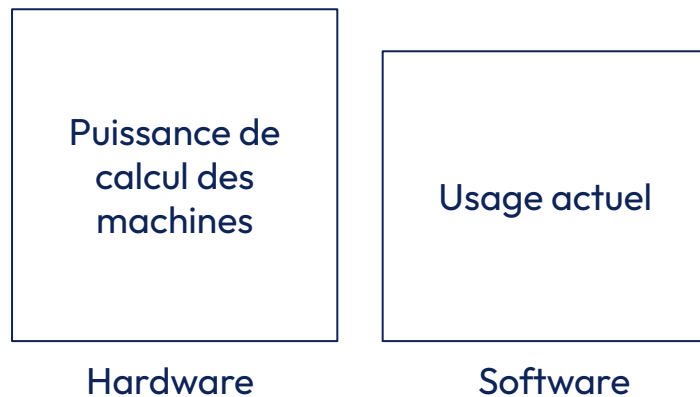
<https://eltoulemonde.fr/>



**Il était une fois la loi de  
Eroom proposée par Tristan  
Nitot**



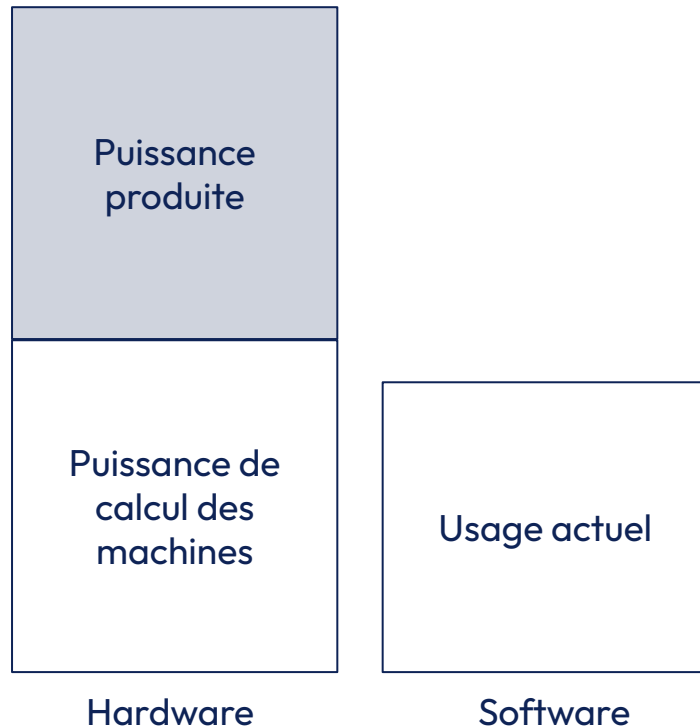
# L'évolution du numérique





# L'évolution du numérique

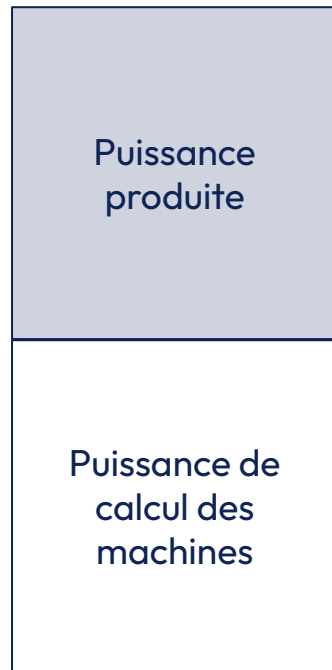
Loi de Moore



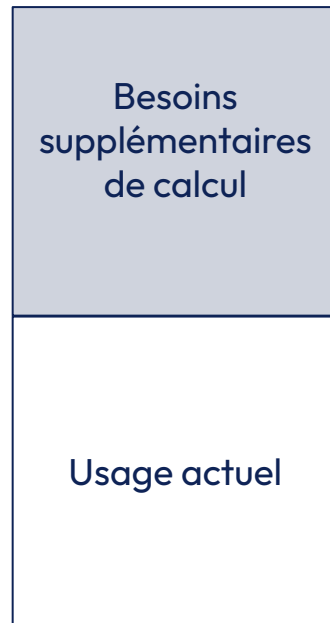


# L'évolution du numérique

Loi de Moore



Hardware

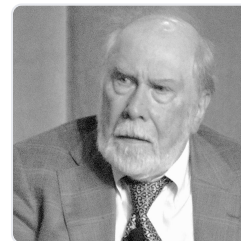


Software







**x Tous les 2 ans**

Loi de Wirth





# Les impacts du numérique en France

%	 Énergie	 GES	 Eau	 Ressources
Fabrication	41%	83%	88%	100%
Utilisation	59%	17%	12%	0%

Source : étude iNum2020, 30 janvier 2021 — <https://www.greenit.fr/impacts-environnementaux-du-numerique-en-france/>



## Une proposition : La loi de Eroom

### Loi d'eroom

Effort Radicalement Organisé  
d'Optimisation en Masse



Optimiser le logiciel  
d'un facteur 2 tous  
les 2 ans

En optimisant le logiciel d'un facteur 2 tous les deux ans, on libère de la puissance informatique avec laquelle on peut inventer de nouveaux usages.

C'est comme la loi de Moore, mais **sans changer le matériel !**

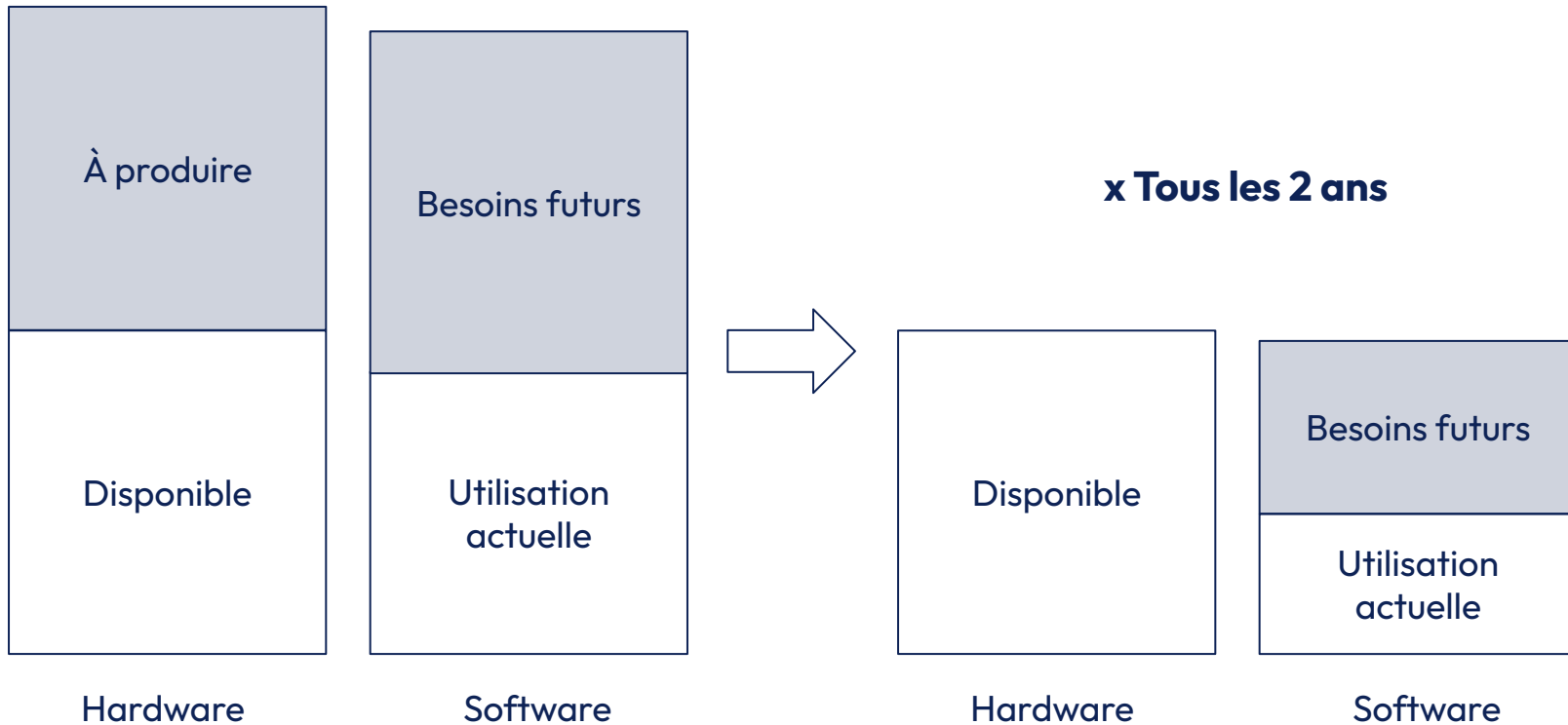




Passer de ça ...

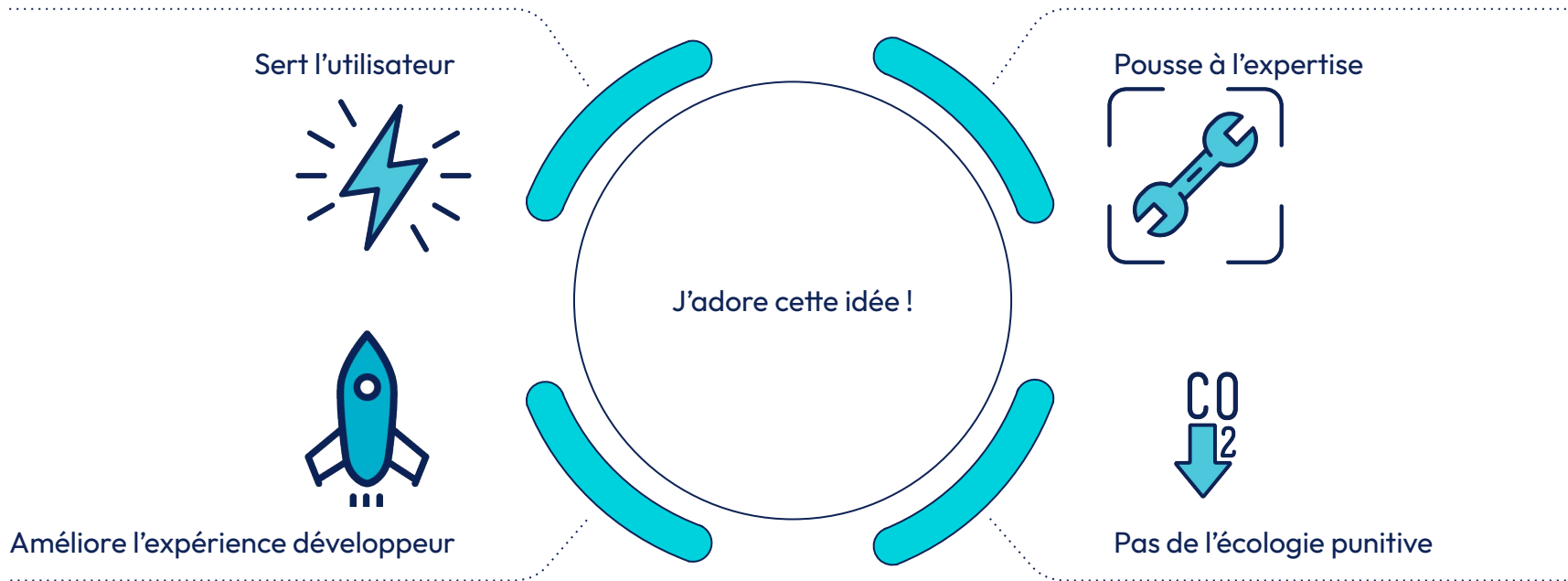
à

... ça





## Pourquoi je vous en parle ?





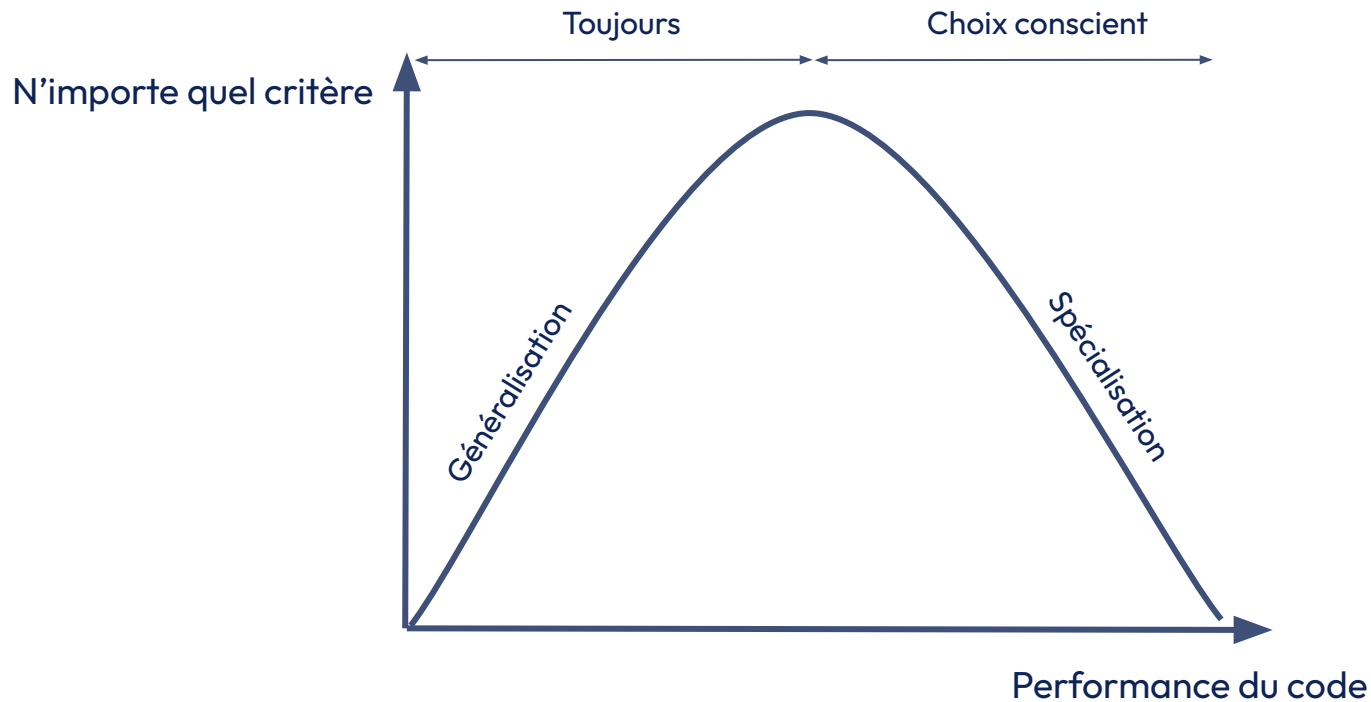
Mon expérience à développer des applications me convainc que

Les **dix premières années de Eroom**, une division par 32 de la puissance nécessaire est **accessible** dans toutes nos organisations.

Ce talk est une illustration de comment faire sur une application.

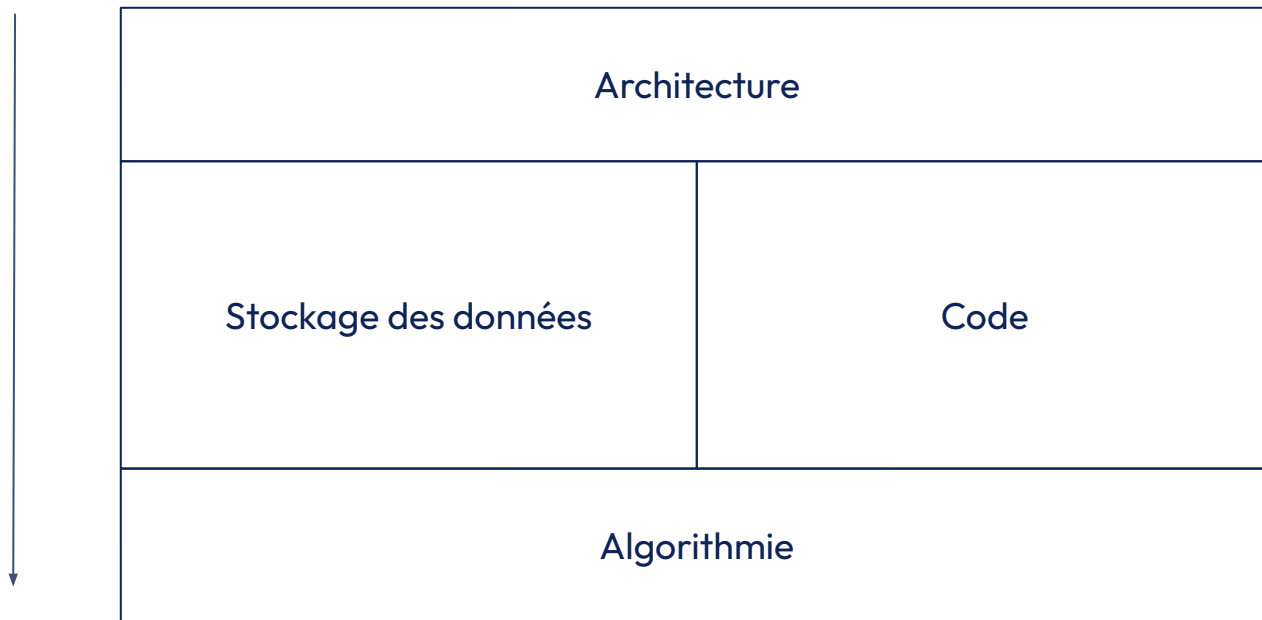


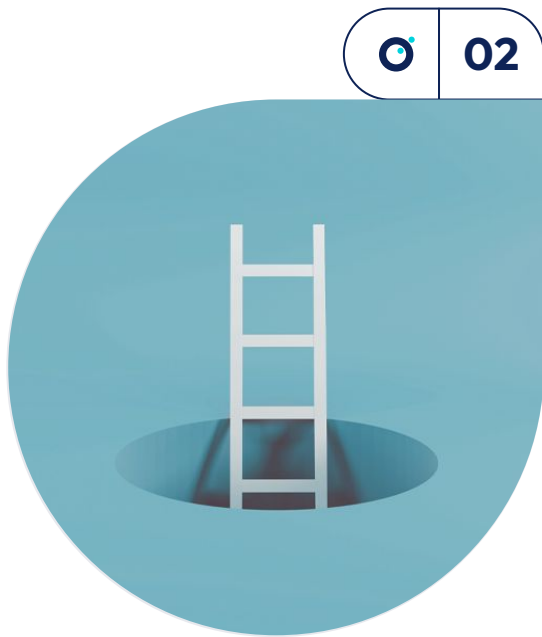
## Proposition : Une grille de lecture pour l'ensemble des optimisations





## 4 volets à creuser





## Afficher des indicateurs

- Parlons d'architecture



# Un exemple pour illustrer

## Statistiques et visualisations de données Covid19

CovidTracker est un outil permettant de suivre l'évolution de l'épidémie à Coronavirus en France et dans le monde. Pour des analyses quotidiennes des chiffres, vous pouvez suivre [@guillaumerozier](#) sur Twitter, ainsi que [@covidtracker\\_fr](#).

### En un coup d'œil

Mise à jour : 2023-03-31

46

BAS ET EN BAISSE • 01/05 (3-3)

#### Taux d'incidence

Nombre de cas par semaine pour 100k habitants. Le seuil d'alerte est 50.

16.82%

TRÈS ÉLEVÉ ET EN BAISSE • 01/05 (3-3)

#### Taux de positivité

Proportion des tests qui sont positifs parmi l'ensemble des tests.

0.86

MODÉRÉ ET EN BAISSE • 31/03

#### Taux de reproduction R

Nombre de personnes contaminées par 1 malade. Au-dessus de  $R=1$ , l'épidémie progresse.

13.8%

BAS ET EN BAISSE • 31/03

#### Tension hospitalière

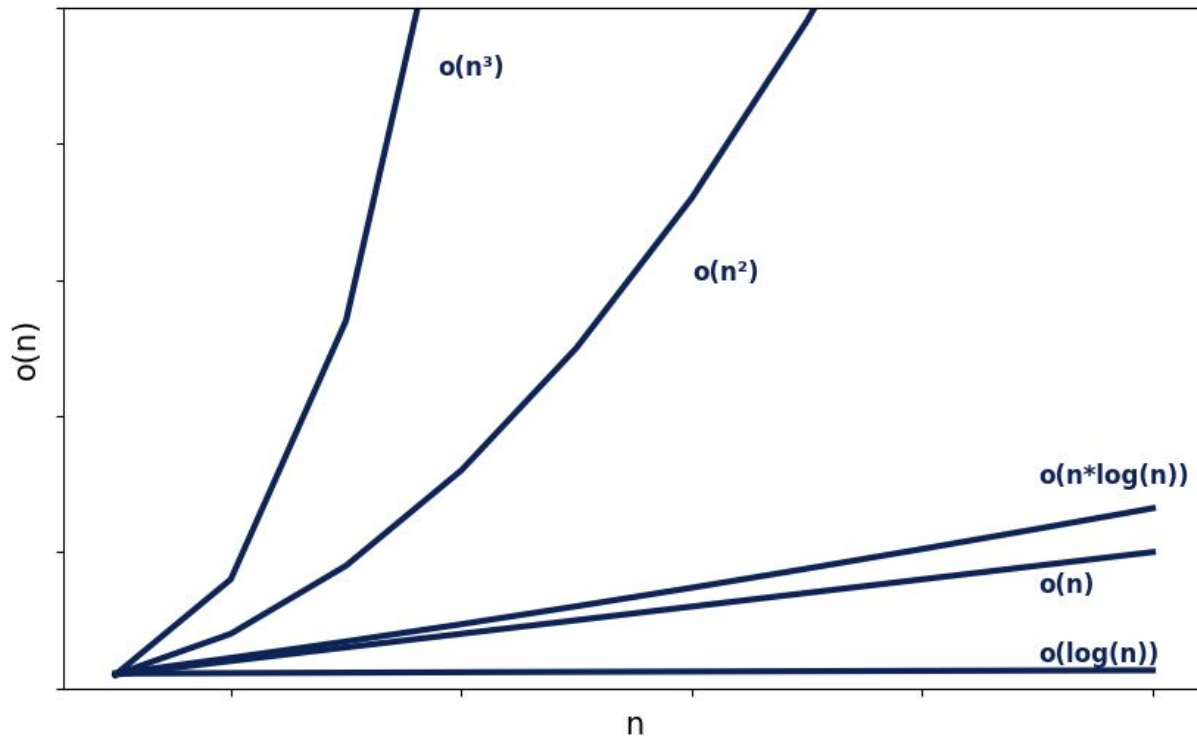
Nombre de lits de soins critiques occupés par les patients Covid19 par rapport au nombre de lits fin 2018.

La tâche à réaliser est de collecter des données depuis un système opérationnel, de calculer des statistiques et de les afficher à l'ensemble des visiteurs.



# Évaluer la complexité algorithmique

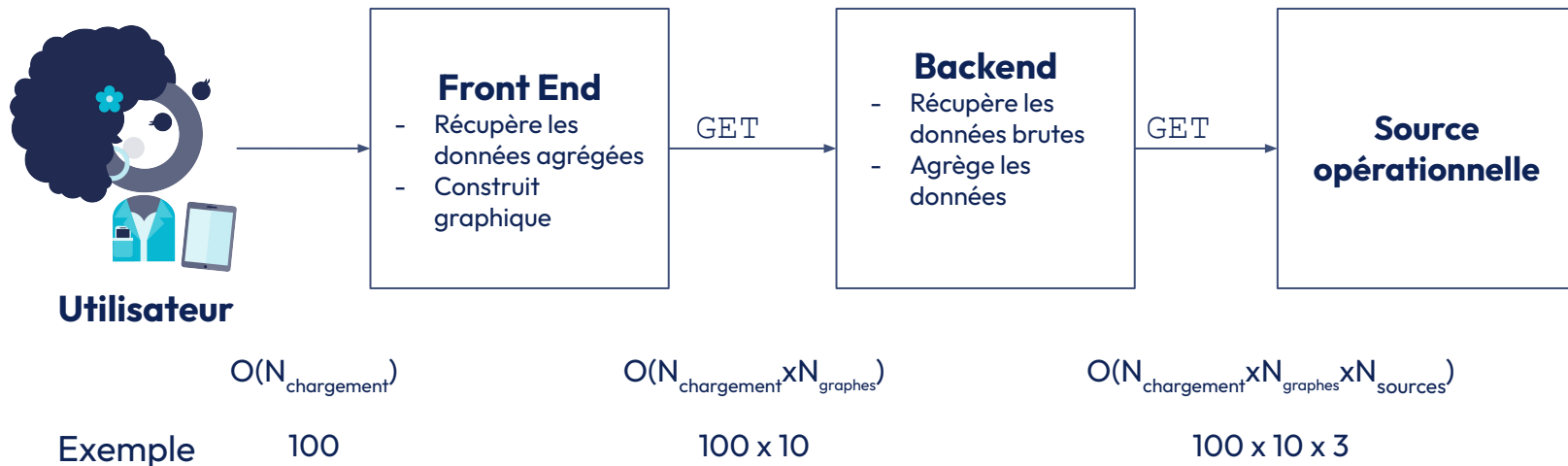
Elle peut être temporelle (~CPU) ou spatiale (~mémoire).





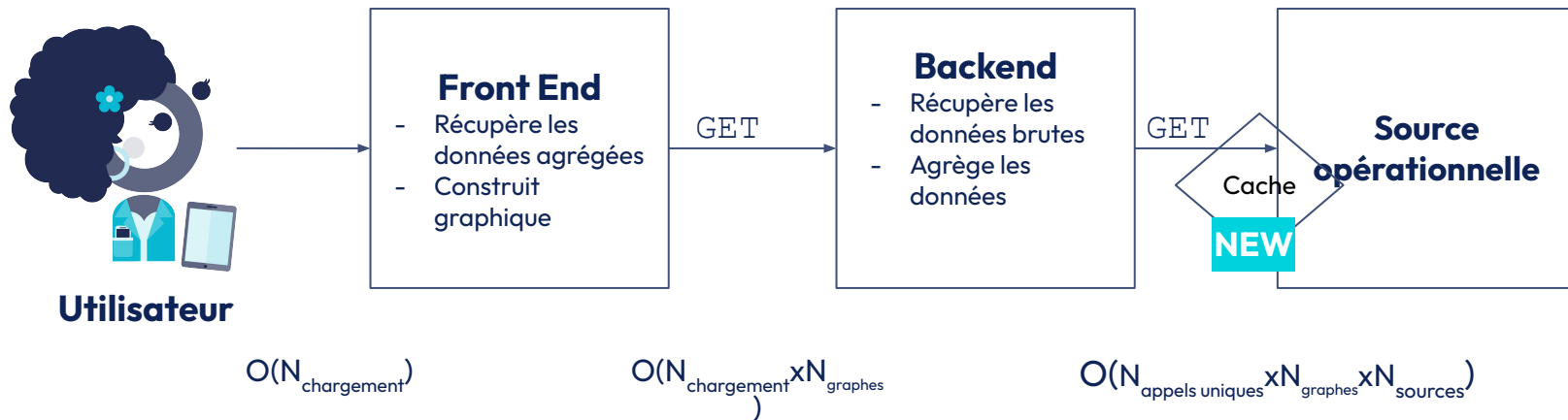


# Architecture minimaliste



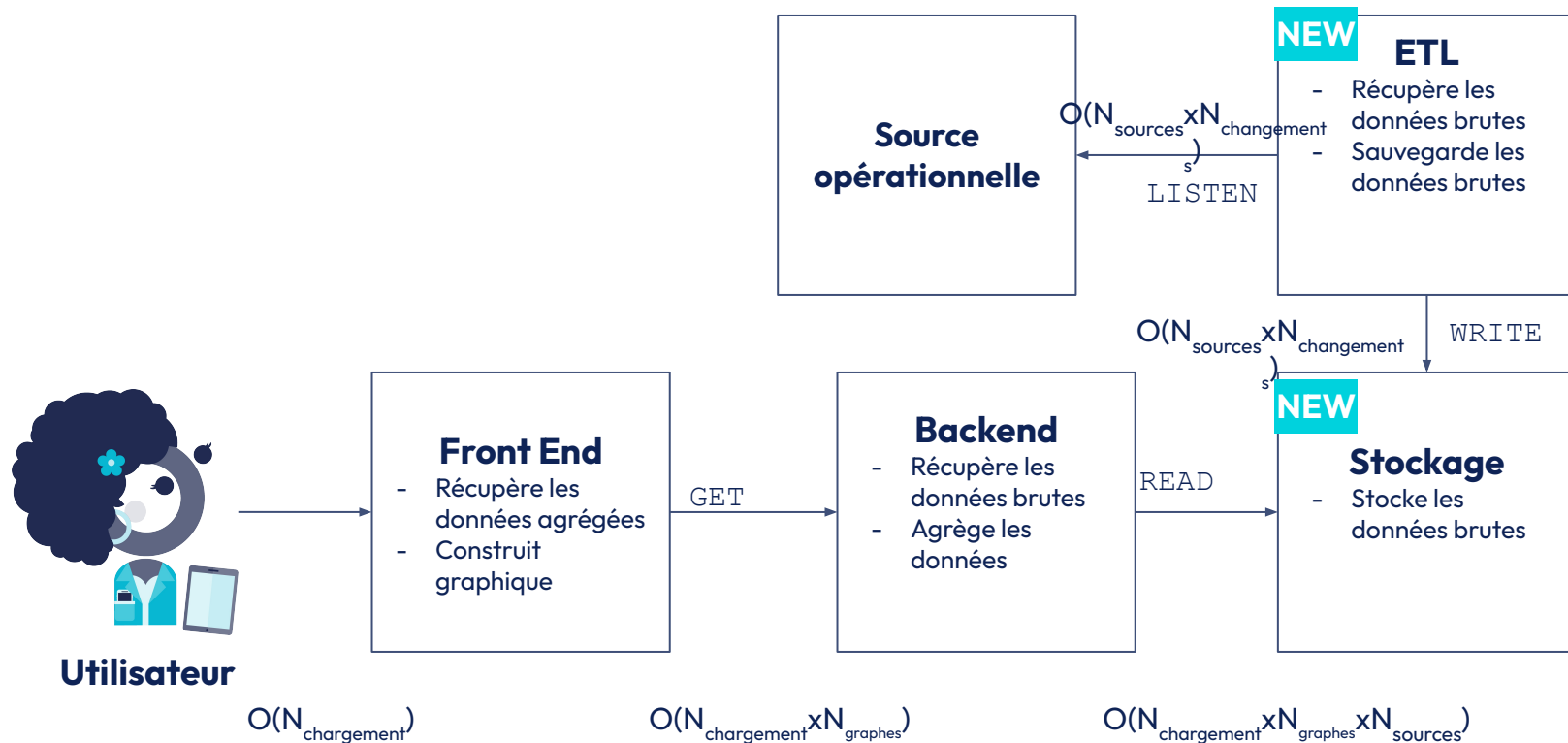


# Architecture avec un cache



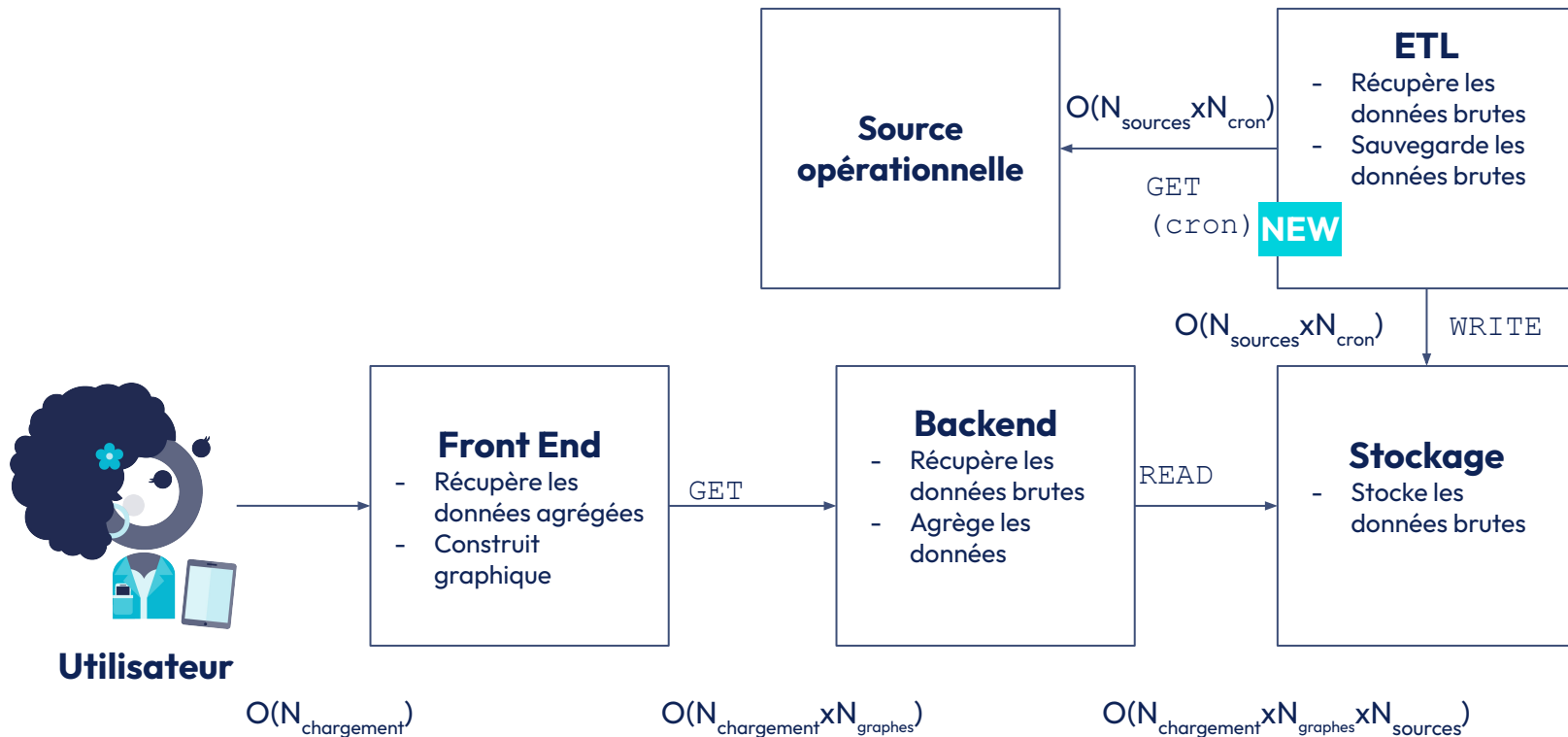


# Architecture découplant système analytiques et opérationnels



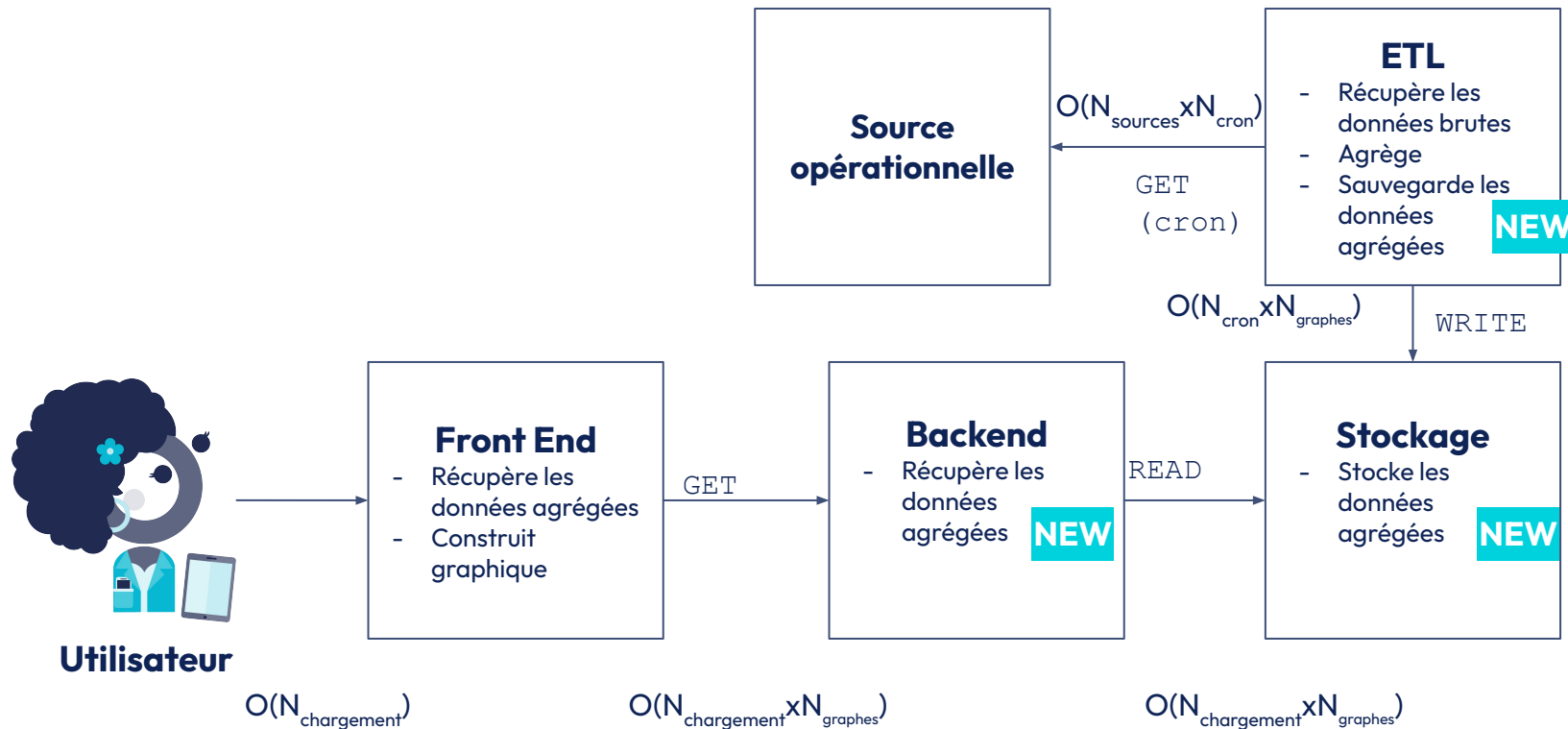


# Architecture challengeant le besoin de temps réel



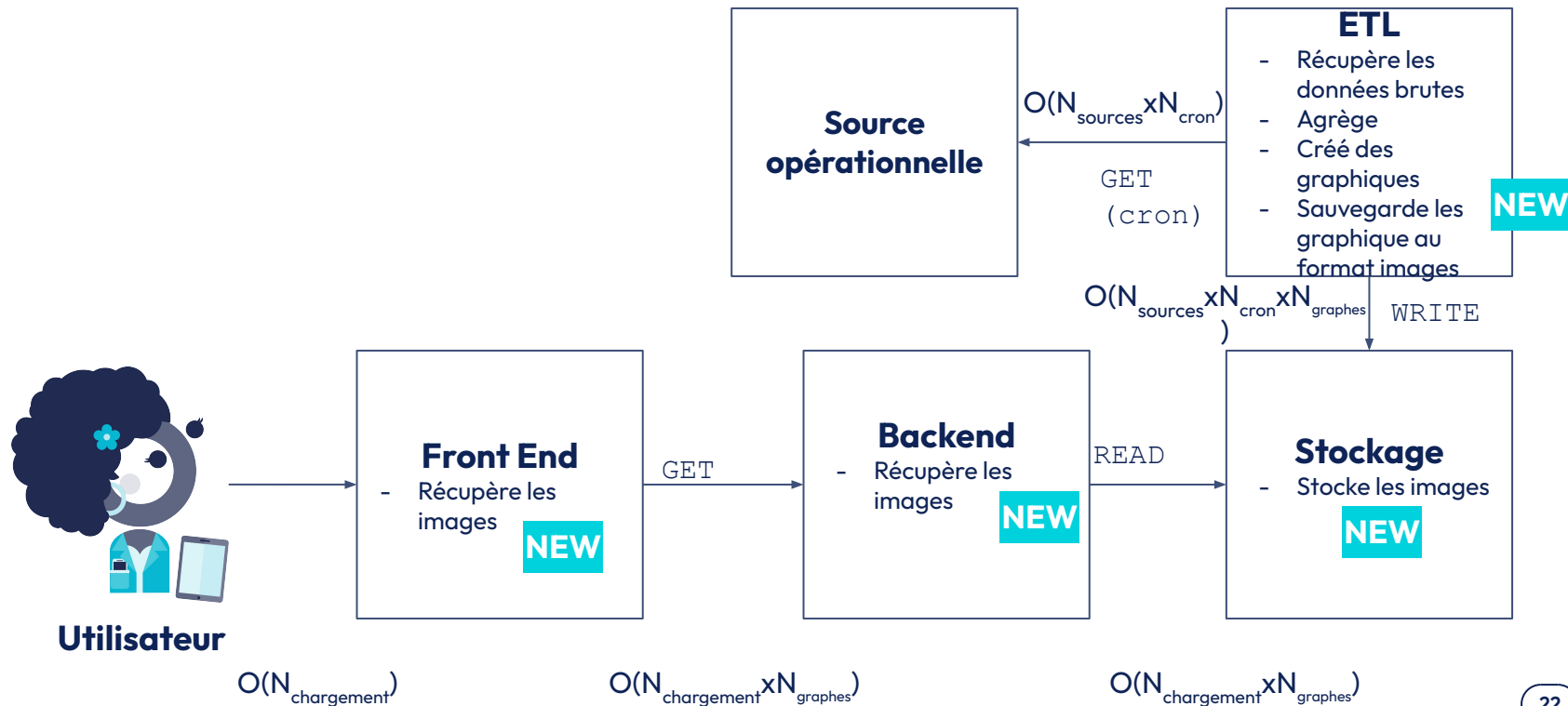


# Architecture redistribuant les rôles Back / ETL





# Architecture redistribuant les rôles FRONT / ETL

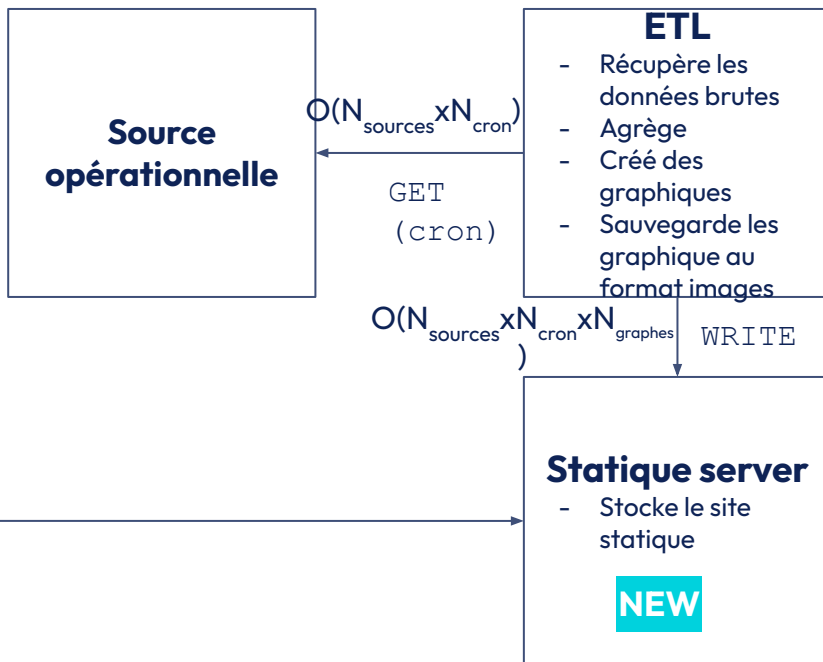




# Architecture site statique



Utilisateur





# Bilan des opérations

Ente ordre de grandeur (grand O)

Architecture	Opérations internes	Requêtes externes
Minimaliste	$N_{\text{chargement}} \times N_{\text{graphes}} \times N_{\text{sources}}$	$N_{\text{chargement}} \times N_{\text{graphes}} \times N_{\text{sources}}$
Avec Cache		$N_{\text{appels uniques}} \times N_{\text{graphes}} \times N_{\text{sources}}$
Découplant (event)		$N_{\text{sources}} \times N_{\text{changements}}$
Découplant (cron)		$N_{\text{sources}} \times N_{\text{cron}}$
Redistribution back / ETL	$N_{\text{chargement}} \times N_{\text{graphes}} + N_{\text{graphes}} \times N_{\text{cron}}$	
Redistribution ETL - Front	$N_{\text{chargement}} \times N_{\text{graphes}} + N_{\text{graphes}} \times N_{\text{cron}}$	
Site statique	$N_{\text{graphes}} \times N_{\text{cron}}$	





# Bilan des opérations avec des chiffres

Ente ordre de grandeur (grand O)

100 utilisateurs / jours, 10 graphes, 3 sources par graphe, 1 update par min, cron 1 x par heure

Architecture	Opérations internes	Requêtes externes
Minimaliste	3 000	3 000
Avec Cache		Entre 30 et 3 000
Découplant (event)		43 200
Découplant (cron)		720
Redistribution back / ETL	1 000 + 240	
Redistribution ETL - Front	1 000 + 240	
Site statique	240	

12 fois moins  
7 ans de Eroom

4 fois moins  
4 ans de Eroom



# Bilan des opérations avec des chiffres

Ente ordre de grandeur (grand O)

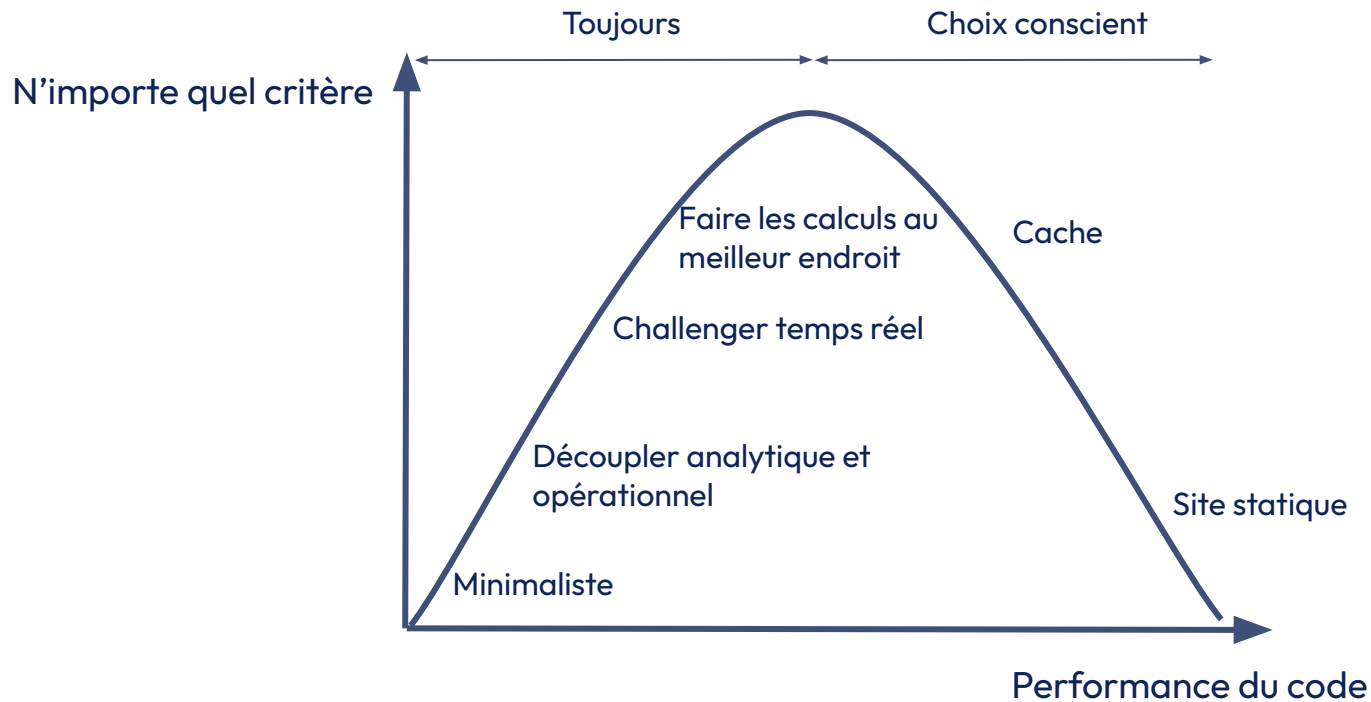
1M utilisateurs / jours, 20 graphes, 3 sources par graphe, 1 update par seconde, cron 1 x par min

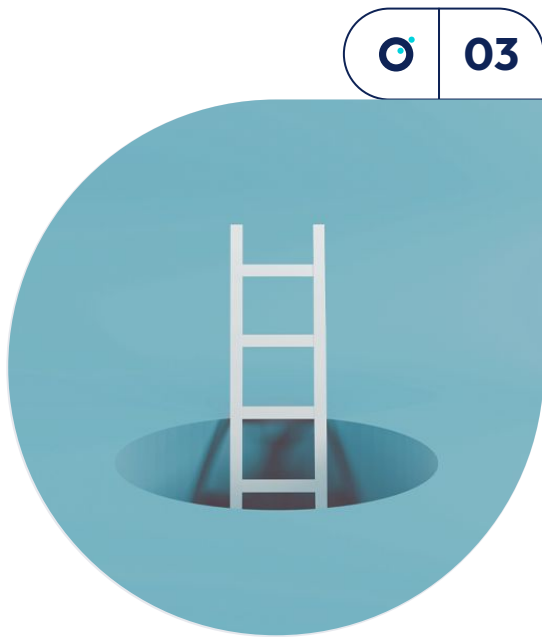
Architecture	Opérations internes	Requêtes externes
Minimaliste	60 000 000	60 000 000
Avec Cache		Entre 60 et 60 000 000
Découplant (event)		5 140 000
Découplant (cron)		86 400
Redistribution back / ETL	20 000 000 + 28 800	
Redistribution ETL - Front	20 000 000 + 28 800	
Site statique	28 800	

2 083 fois moins  
22 ans de Eroom

694 fois moins  
9 ans de Eroom

# Les pratiques sur la grille de performance vs ...



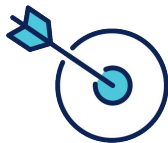


**Calculer des statistiques sur  
des grosses données !**

- Parlons code



## Un exemple pour illustrer



La tâche à réaliser est :

1. Calculer la moyenne de “numeric”
2. Calculer la proportion de “a”

<b>numeric</b> (float32)	<b>letter</b> (category)	<b>partition</b> (int16)
0.99828804	b	54
0.7377946	c	31
0.7337601	c	9
0.29893994	b	25
0.2920279	b	75
0.22539395	b	26
0.99200934	a	89

x 50 millions de lignes



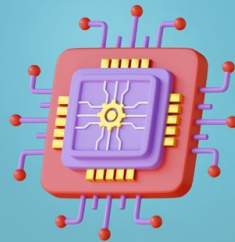
## 3 indicateurs à suivre

Max mémoire utilisée  
Avec memray



Temps de calcul  
Comme proxy de l'usage CPU

Plus gros objet  
Avec sys.getsizeof





# Version développeur qui ne connaît pas les packages data

```
1 def row_csv_approach(csv_file_path: str):
2     total sum, n_lines, a_count = 0
3     with open(csv_file_path, newline='') as csvfile:
4         reader = csv.reader(csvfile)
5         next(reader) # Skip header row
6         for row in reader:
7             total sum += float(row[0])
8             n_lines += 1
9             a_count += (row[1] == 'a')
10    return total_sum / n_lines, a_count / n_lines
```

Pic mémoire	Plus gros objet	Temps
48 Mb	104 b	82s



## Avantages :

- Pas de framework,
- ~Même code dans tous les langages
- Parcimonieux en RAM



## Inconvénients :

- Très lent



## Version naïve

```
1 def naive_approach(csv_path: str):  
2     df = pd.read_csv(csv_path)  
3     mean = df["numeric"].mean()  
4     share_of_a = sum(df["letter"] == "a") / df.shape[0]  
5     return mean, share_of_a
```

Pic mémoire	Plus gros objet	Temps
3 100 Mb	3 700 Mb	43s

Valeurs de référence !



Avantages :

- ◉ Simple et efficace !
- ◉ On a tous déjà écrit cela
- ◉ Code très découplé



Inconvénients :

- ◉ C'est coûteux en mémoire et en temps





## Version frugale

```
1 def frugal_approach(csv_path: str):  
2     df = pd.read_csv(csv_path, usecols=["numeric", "letter"])  
3     mean = df["numeric"].mean()  
4     share_of_a = sum(df["letter"] == "a") / df.shape[0]  
5     return mean, share_of_a
```

Pic mémoire	Plus gros objet	Temps
2 700 Mb	3 300 Mb	38s
-13%	-11%	-12%



Avantages :

- Gain facile de mémoire et de temps de calcul



Inconvénients :

- ?



```
1 def parquet_approach(parquet_path: str):
2     df = pd.read_parquet(parquet_path, columns=["numeric", "letter"])
3     mean = df["numeric"].mean()
4     share_of_a = sum(df["letter"] == "a") / df.shape[0]
5     return mean, share_of_a
```

Pic mémoire	Plus gros objet	Temps
1 400 Mb	250 Mb	16s
-55%	-93%	-63%



## Avantages :

- 3 fois moins d'espace disque pris
- Pas d'inférence des types
- Temps de lecture / écriture accélérée



## Inconvénients :

- Ne peut plus ouvrir les données dans Excel



## Version chunk

```
1 def chunk_approach(parquet_path: str):
2     all_partitions = glob.glob(f"{parquet_path}/*")
3     total = 0
4     n_lines = 0
5     number_of_a = 0
6     for partition in all_partitions:
7         chunk = pd.read_parquet(partition, columns=["numeric", "letter"])
8         total += chunk["numeric"].sum()
9         n_lines += chunk.shape[0]
10        number_of_a += (chunk["letter"] == "a").sum()
11    return total / n_lines, number_of_a / n_lines
```

Pic mémoire	Plus gros objet	Temps
168 Mb	2,53 Mb	20s
-95%	-99,9%	-53%



### Avantages :

- Réduction drastique de l'usage mémoire



### Inconvénients :

- Le code est plus complexe
- Couplage plus fort entre les 2 calculs, la lecture



# Version parallélisée

```
1 def sum_and_count(file_path: str):
2     df = pd.read_parquet(file_path, columns=["numeric", 'letter'])
3     return df["numeric"].sum(), df.shape[0], (df["letter"] == "a").sum()
4
5
6 def parallel_mean(file_path: str):
7     all_partitions = glob.glob(f"{file_path}/*")
8     n_core = 8
9     with Pool(n_core) as p:
10         result = p.map(sum_and_count, all_partitions)
11     return sum([r[0] for r in result]) / sum([r[1] for r in result]),
12         sum([r[2] for r in result]) / sum([r[1] for r in result])
```

Pic mémoire	Plus gros objet	Temps
49.4 Mib	2,53 Mb * 8	9,5s
-98%	-99,4%	-78%



## Avantages :

- Plus rapide



## Inconvénients :

- Le code est plus complexe
- Il y a un couplage plus fort entre les 2 calculs, la lecture
- Gourmand en CPU



# Version pyspark

```
1 def pyspark_approach(parquet path: str):
2     spark = SparkSession.builder.appName("Code Example").getOrCreate()
3     df = spark.read.parquet(parquet path)
4     average_value = df.selectExpr("avg(numeric)").collect()[0][0]
5     total_rows = df.count()
6     rows_equal_a = df.filter(col("letter") == "a").count()
7     share_of_a = rows_equal_a / total_rows
8     spark.stop()
9     return average_value, share_of_a
```

Pic mémoire	Plus gros objet	Temps
?	?	35,4s
		-18%



## Avantages :

- Scale très bien à des données trop grosses pour un disque



## Inconvénients :

- Un autre style avec des compétences différentes
- Plus difficile à observer
- Le temps est consacré à de la "plomberie"



# Version polars

```
1 def polars_approach(parquet_path: str):
2     df = pl.read_parquet(parquet_path, columns=["letter", "numeric"])
3     average_numeric = df['numeric'].mean()
4     proportion_a = df.filter(pl.col('letter') == 'a').height / df.height
5     return average_numeric, proportion_a
```

Pic mémoire	Plus gros objet	Temps
?	381 Mb	1,03s
	-90%	-98%



## Avantages :

- Permet d'intégrer du rust dans du python
- ça pulse



## Inconvénients :

- Un autre style avec des compétences différentes
- Framework moins mature

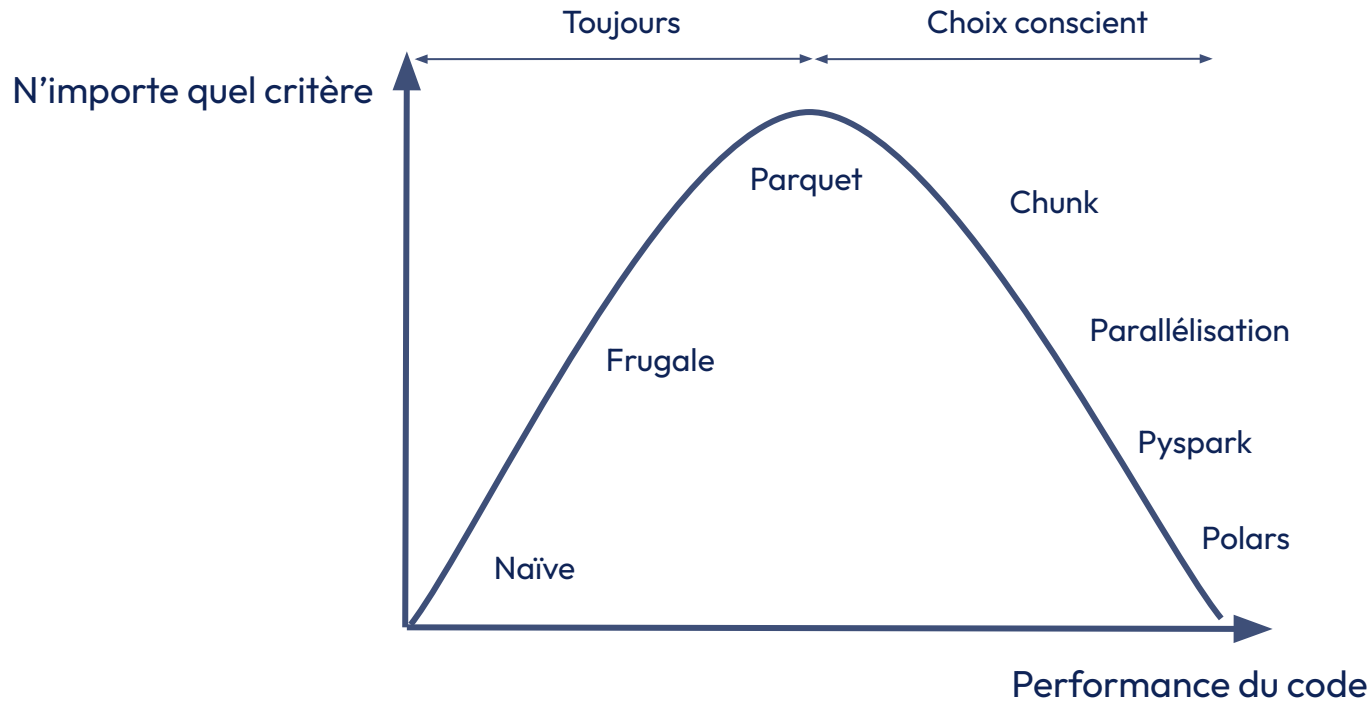


## En résumé, les gains de performance

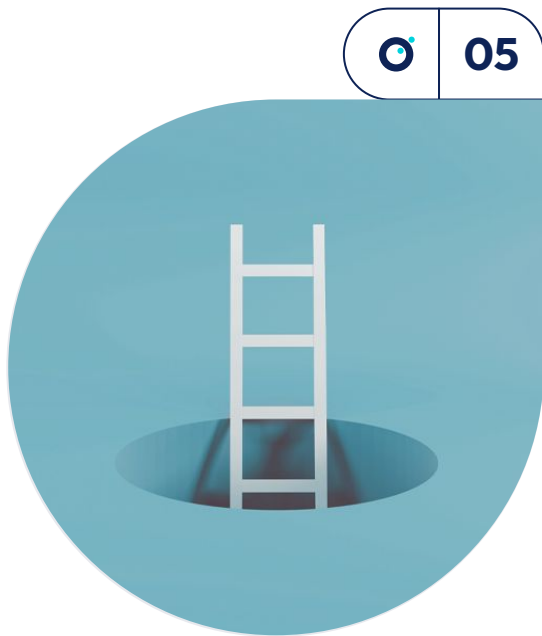
Version	Pic mémoire	Plus gros objet	Temps de calcul	Quand ?	Années Eroom
Dev	48 Mb	104 b	82s	Jamais ?	
Naïve	3 100 Mb	3 700 Mb	43s	Prototype / minidonnées	Référence
Frugale	-13%	-11%	-12%	Toujours	0,35
Parquet	-55%	-93%	-63%	Toujours	2
Chunk	-95%	-99,9%	-53%	Données > Mémoire	8
Parallélisé	-98%	-99,4%	-78%	Calculs lourds & CPU disponible	12
Pyspark	?	?	-18%	>> 10Gb / jours	?
Polars	?	-90%	-98%	Si besoin vraiment spécifique	11



## Les différentes optimisations sur la grille performance vs ...





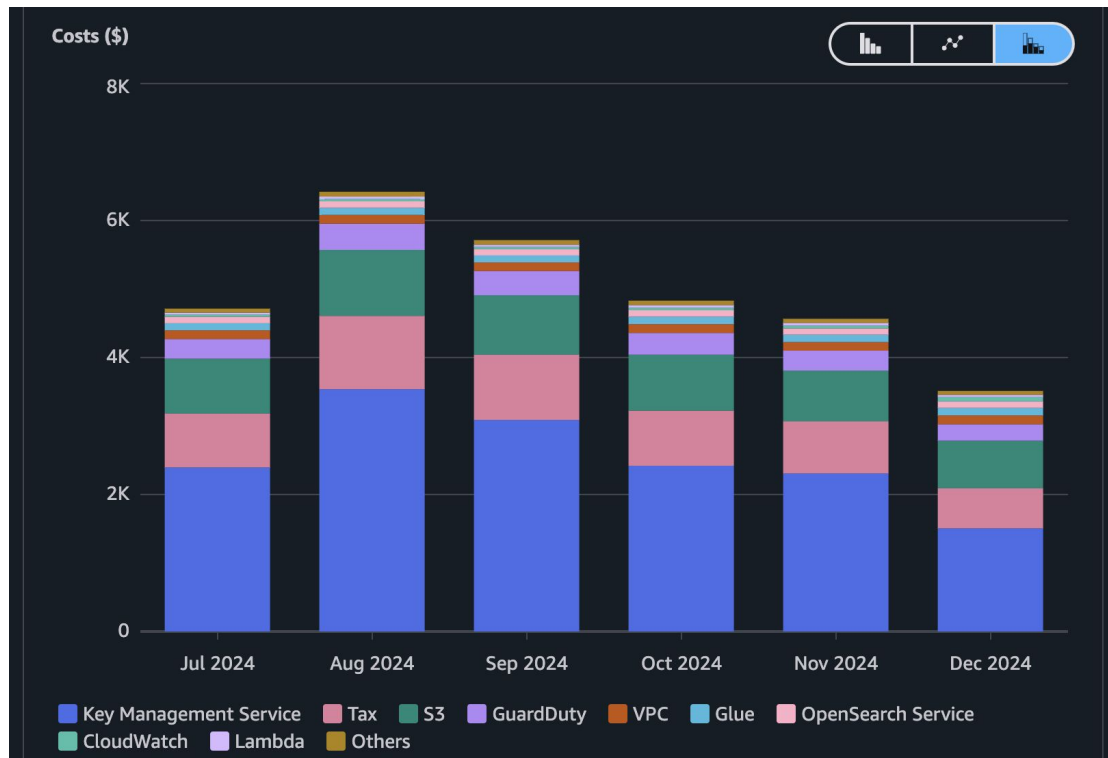


**Des exemples de la vraie vie !**



# Explorons la console de coût d'une data plateforme

Catégorie bonne pratique



- KMS un gestionnaire de secret
- S3 du stockage

Morale : Explorez vos coûts !



# En BI, tu as fait du mauvais boulot quand c'est lent !

Catégorie bonne pratique

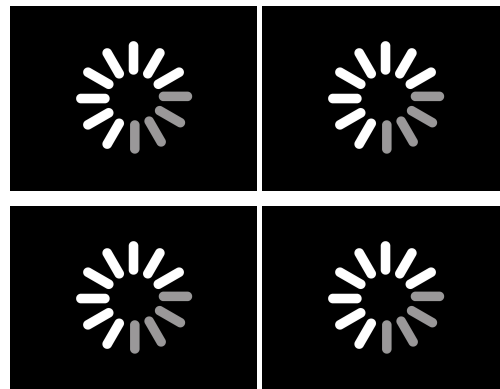
## Avant

Côté data engineering :

- Insert 50 millions de lignes dans PowerBI
- Le lendemain, 51 millions de lignes
- Le sur-lendemain, 52 millions de lignes

Côté BI :

- Calculer des indicateurs dessus



## Après

Côté Data engineering :

- Calcul les agrégats et stock les résultat
- Ingère (éventuellement) en incrémental dans PowerBI

Côté BI :

- Affiche l'indicateur



# Un job d'agrégation de time series

Catégorie bonnes pratiques !

## Contexte

Sur des times series assez volumineuses, il faut calculer des indicateurs toutes les minutes.

## Après

Un script python incrémental idempotent qui réalise les calculs de la dernière minute et l'append au fichier parquet.  
Le job tourne en quelques secondes.

## Avant

Un job spark qui calcule les indicateurs de 00:00 à maintenant.  
Écrase la journée d'aujourd'hui.  
Le job ne tourne pas assez vite en fin de journée.

## Morale

Rendre le code efficient plutôt  
que de mettre plus de  
puissance





# Optimisation de la concaténation

Catégorie bonne pratique

Avant

```
def a super function(df: pd.DataFrame) -> pd.DataFrame:
    result df = pd.DataFrame(...)
    for ... in ...:
        sub df = ...
        result_df = pd.concat([result_df, sub_df],axis=1)

    return result_df
```

Contexte :  
Vu au fin fond d'un job de  
data eng.

Après

```
def a super function(df: pd.DataFrame) -> pd.DataFrame:
    list sub df = []
    for ... in ...:
        sub df = ...
        list_sub_df.append(sub_df)

    return pd.concat(list_sub_df, axis=1)
```

Résultat :  
Gain de 30% de temps de  
calculs dans ce cas

# Optimisation de la gestion des connections

Catégorie choix conscient

## Contexte

```
list df = []
for file in big list of files:
    df = blob_handler.read_bronze_file(file_path=file)
```

## Avant

```
class BlobHandler(...):

def read_bronze_file(self, file_path: str) ->
pd.DataFrame:
    return pd.read_json(file_path,
        storage_options={"anon": False})
```

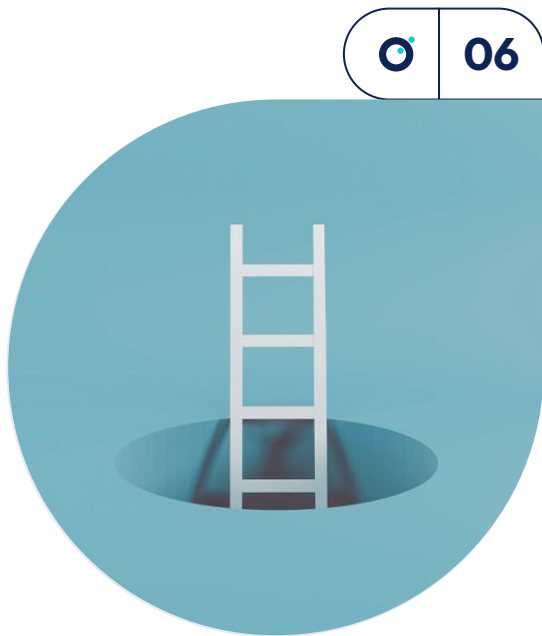
## Après

```
class BlobHandler(...):

def get_container_client(self) -> ContainerClient:
    if self._container_client is None:
        self._container_client = ContainerClient(...)
    return self._container_client

def read_bronze_file(self, file_path: str) ->
pd.DataFrame:
    container_client = self.get_container_client()
    return pd.read_json(BytesIO(
        container_client.download_blob(file_path).readall(
        )))
```

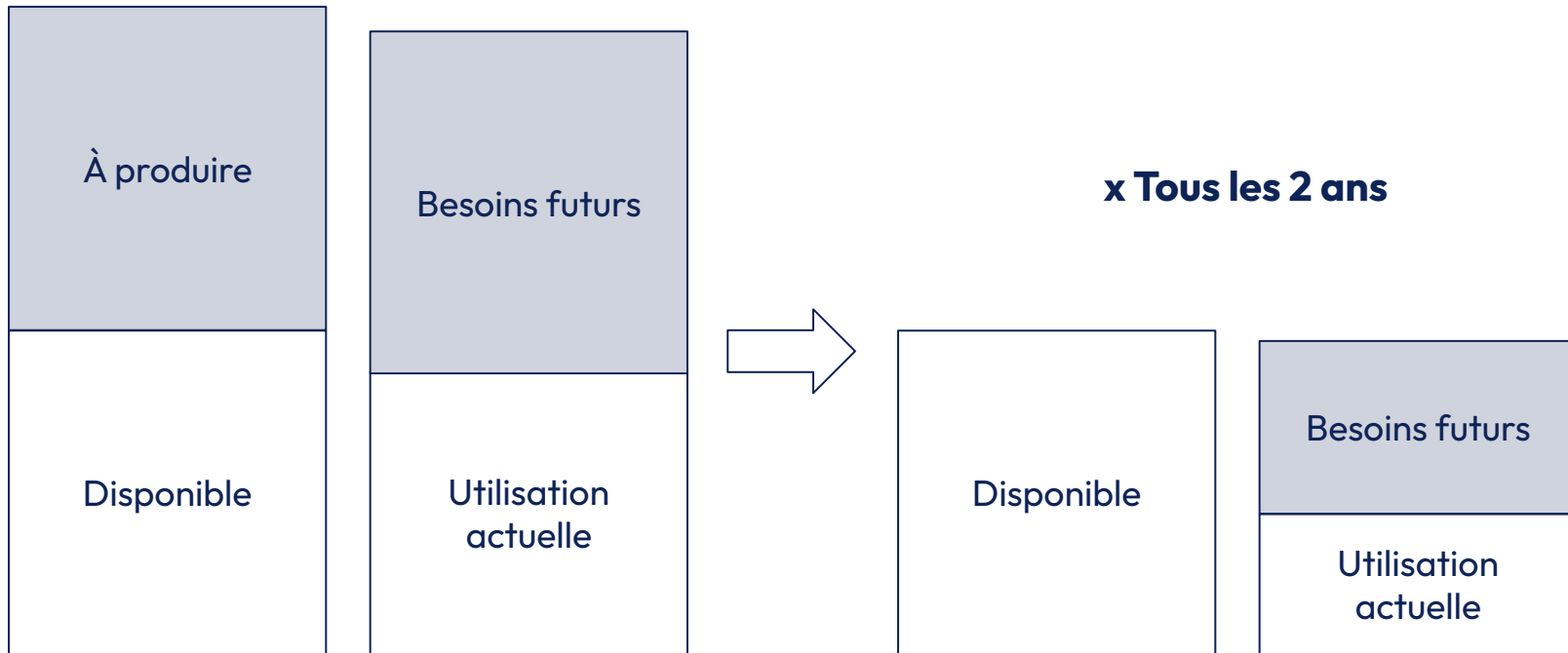
Dans le cas particulier de très nombreuses petite interaction avec le stockage, cacher la connection divise par 2 le temps de calcul



## Conclusion



## Rappel de ce que l'on veut faire





# À l'échelle d'une application, c'est possible



- ◉ L'architecture pour faire les bons calculs
- ◉ Le stockage pour stocker, modifier, récupérer les données efficacement
- ◉ Le code pour faire les choses efficacement
- ◉ L'algorithmie pour utiliser les maths au service de notre problème



## À l'échelle de l'organisation...

### Reprendre l'existant

- ◉ Appliquer la loi de Pareto : trouver les applications, les fonctionnalités les plus gourmandes en ressources
- ◉ Les refactorer
- ◉ En tirer des best practice
- ◉ Identifier des besoins d'expertise et monter une cellule, contribuer à des librairies open sources...

### Améliorer le futur

- ◉ Mettre en place les bonnes pratiques dès maintenant
- ◉ Penser aux refactoring de performance futurs



## Et moi développeur, DE, DA, DS, etc ?

- ◉ Ce qui ne se mesure pas ne s'améliore pas... J'apprends à profiler
- ◉ Je prends du recul sur la gourmandise en CPU / RAM de ce que j'écris
- ◉ Je regarde les nouveaux outils / framework sous l'angle parcimonieux



# MERCI

Convaincu ? On en parle ?  
Pas convaincu ? On en parle ?



Linked'in

<https://eltoulemonde.fr/>



*There  
is  
a Better  
Way*