

**authors:**

- S.LAVAZAIS
- D.SIX

**Sources:**

- [Wikipedia - software performance testing](#)

# LES TESTS DE PERFORMANCE



# INTRODUCTION - SOMMAIRE

1. Les tests performance, qu'est-ce que c'est ?
2. Quel type de test pratiquer?
3. Java Microbenchmark Harness
4. Taurus

## Speaker notes

Pour cette présentation, nous allons voir grossièrement ce que sont les tests de performance, et nous allons faire deux focus sur un sous-domaine de test qui sont le micro benchmark et les tests de charge.

- Le focus sur le micro-benchmark se fera avec Java Microbenchmark Harness (JMH) et sera présenté par Douglas Six.
- Le focus sur les tests de charge se fera avec Taurus et sera présenté par Sylvain Lavazais.

**Mais tout d'abord, parlons des tests de performance...**

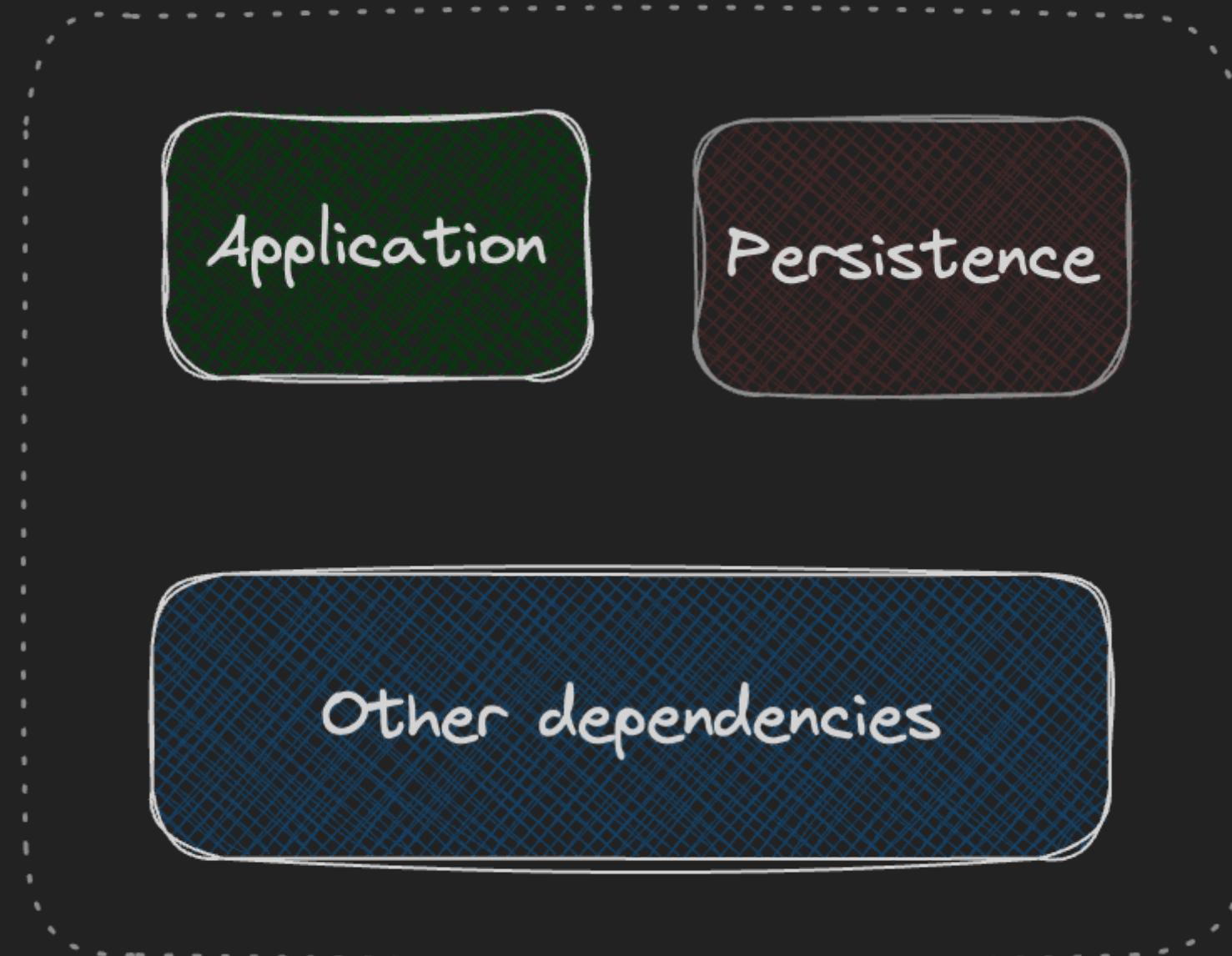
1. Les testes de performance, qu'est-ce que c'est ?
2. Quel type de test pratiquer?
  1. test en isolement
  2. test de charge
  3. test en stress
  4. test de pic
  5. test au limites
3. Tests en isolement avec Java Microbenchmark Harness
4. Tests de charge avec Taurus
  1. Demo de charge
5. Merci !

Les tests de performances, c'est la capacité à mettre une application et toutes ses dépendances dans un contexte virtuel ou réel et observer comment ces composants se comportent dans des conditions de stress / charge, etc.

On peut ainsi en déterminer des axes d'amélioration.

# QU'EST-CE QUE C'EST ?

Test Bench - Context





Voici les principaux métriques qu'un test de performance évalue :

- Le temps de réponse global (sur le pourcentage de toutes les réponses) et à un instant T.
- La quantité de ressources utilisées
- Mesure de la capacité d'adaptation / de la scalabilité

# QUEL BÉNÉFICE EN TIRER ?

temps de réponse    les ressources utilisées    l'adaptation



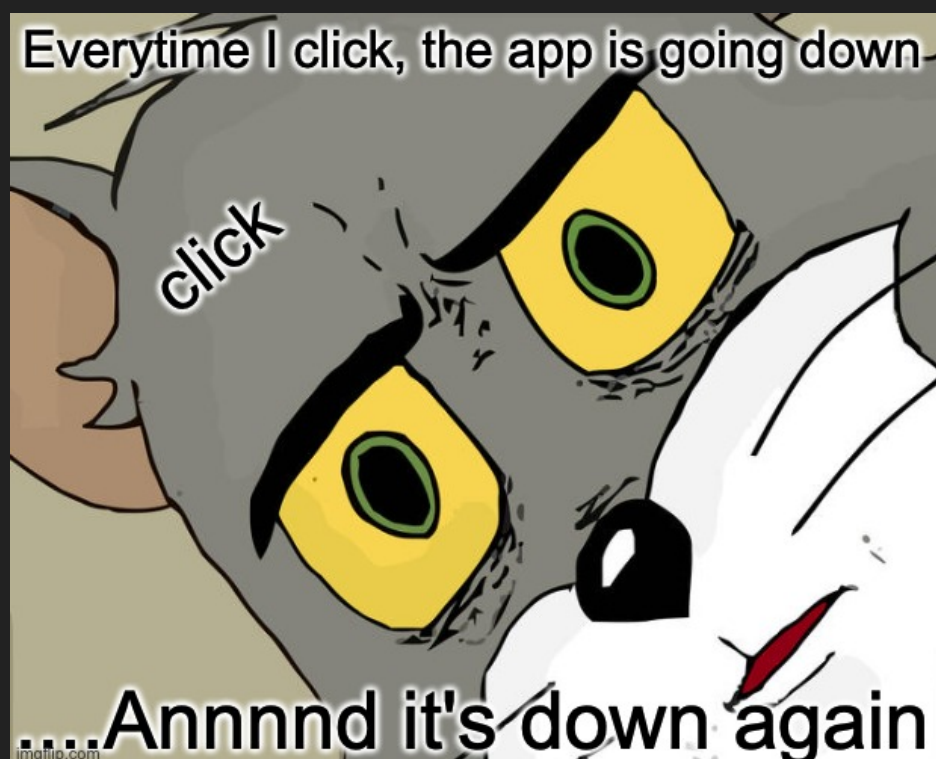


Les tests de performance permet également de :

- De prévenir les coupures dû à la charge
- De détecter les bugs non couverts / les goulots d'étranglement
- D'atténuer les failles de sécurités

# QUEL BÉNÉFICE EN TIRER ?

les coupûres



bug / bottleneck



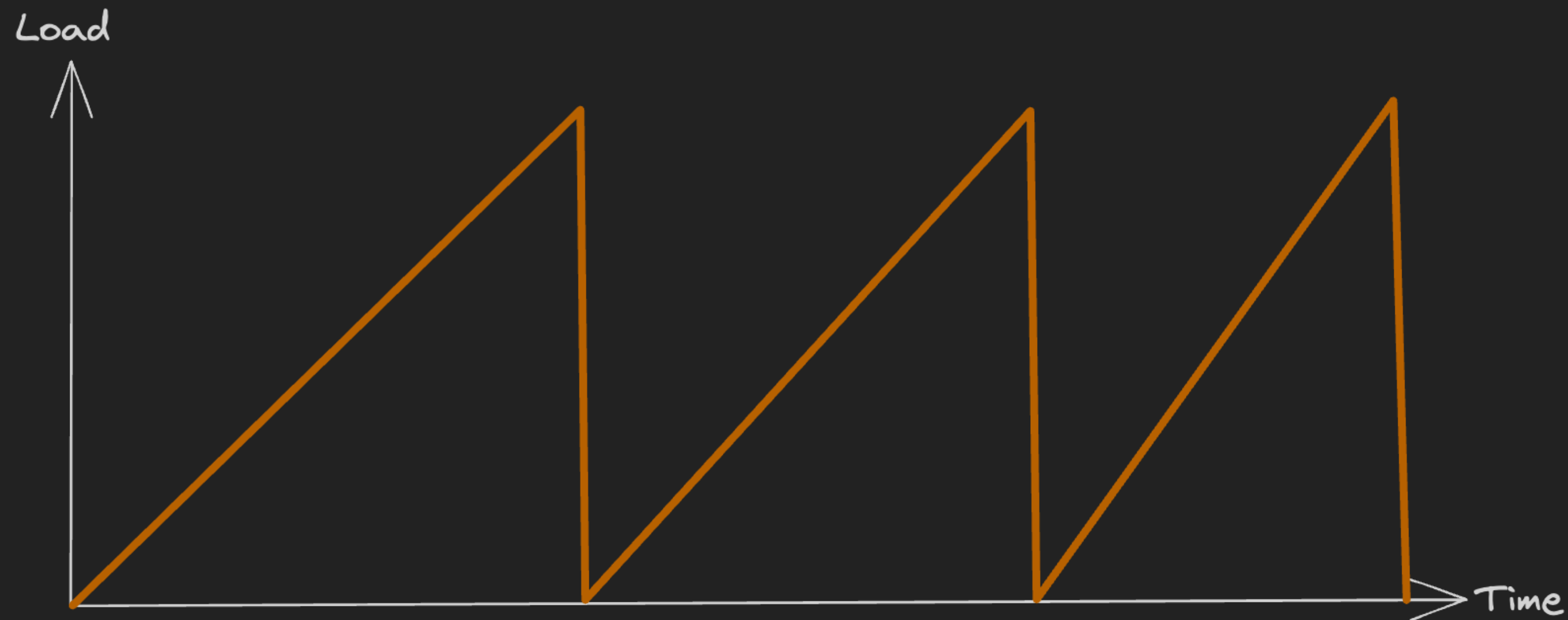
la sécurité



## Douglas

Pour ce type de test on va tester un sous ensemble de l'application finale, par exemple la fonctionnalité d'ajout au panier, mais sans la partie navigation dans le site, l'authent ou le paiement... Le test sera réalisé en pratiquant une répétition d'exécutions.

# TEST EN ISOLATION



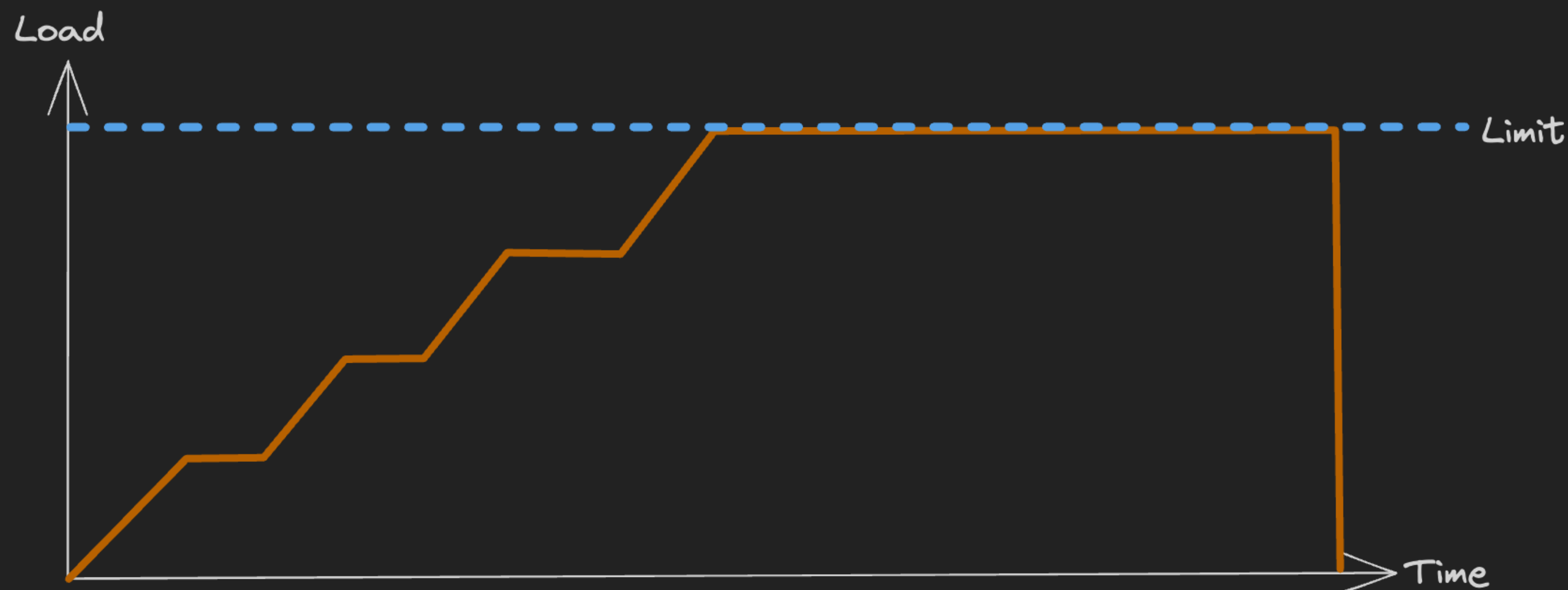
Sylvain

Le test de charge est la forme la plus simple et connu pour tester une application. L'objectif est de vérifier si l'application est capable de gérer les limites de temps de réponse / consommation ressources qui ont été décider préalablement (par exemple au travers d'un SLA, Service Level Agreements) L'infrastructure est également sous-monitoring durant cette phase de test.

Ce type de test peut être utilisé comme un test de qualité d'une release à une autre, ainsi qu'un objectif à maintenir.

Pour aller plus loin, il est possible de prolonger la durée afin d'obtenir un test d'endurance (pour détecter les problèmes de fuite mémoire)

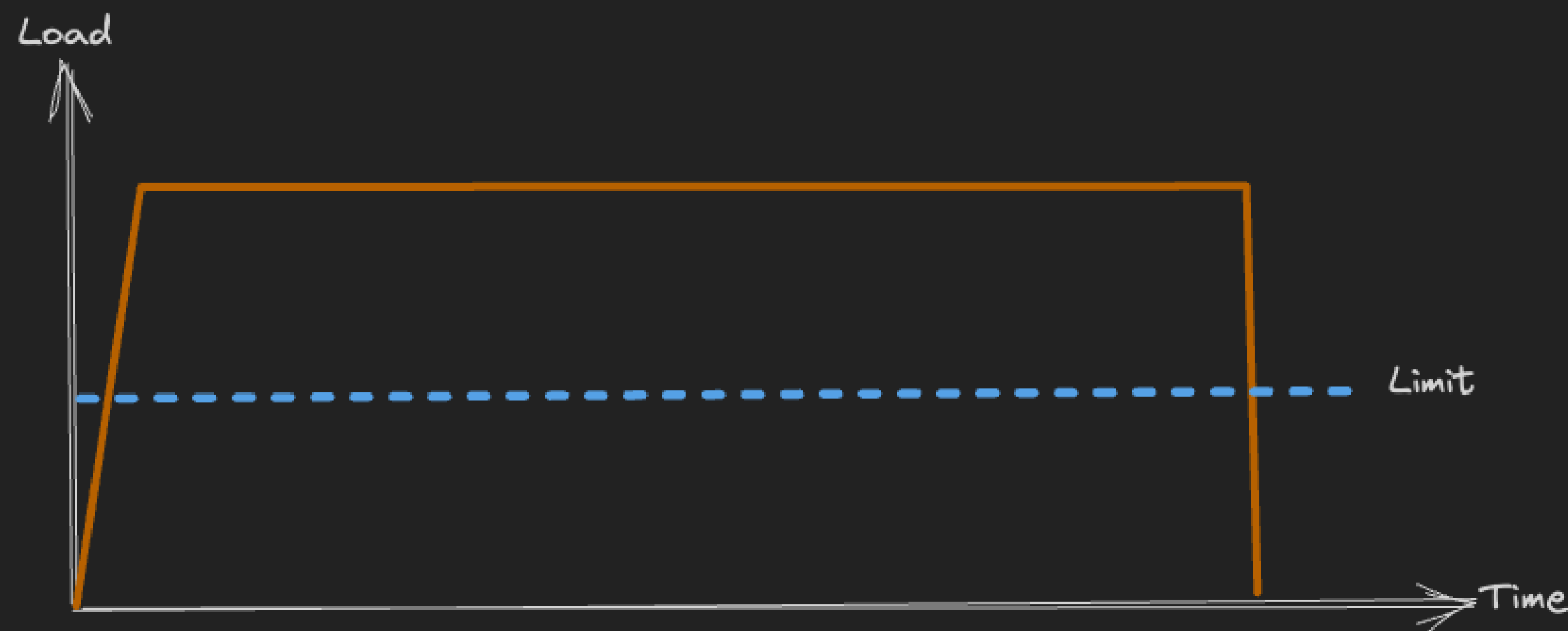
# TEST DE CHARGE



## Douglas

Ce type de test vise à comprendre comment l'application va se comporter face à une charge plus importante que le cas nominal (x2 à x4) en utilisant la même infra que celle ciblée avec une montée en charge rapide. On identifie ainsi les endroits de l'application où se produiront les premières erreurs, ainsi que les temps de réponse en dehors des exigences.

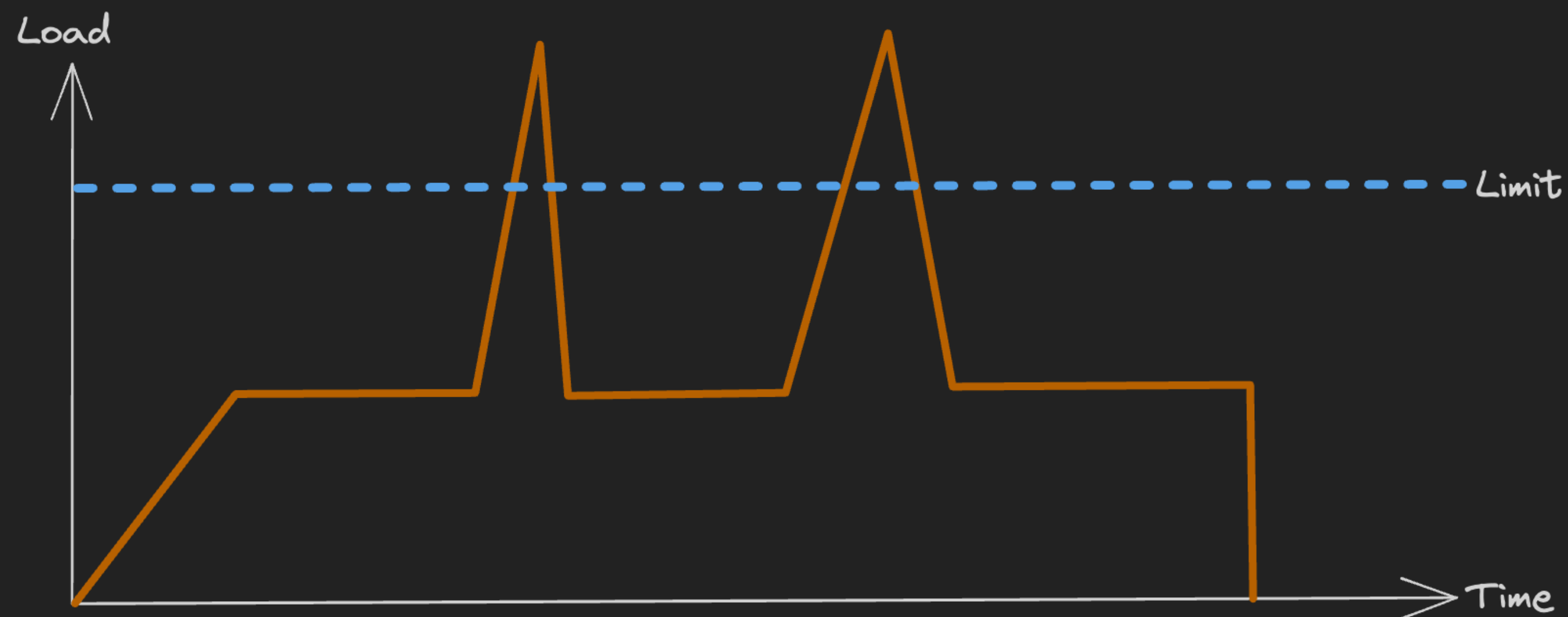
# TEST EN STRESS (SURCHARGE)





Le test de pic vise à déterminer les problèmes de performance quand un changement de contexte se produit sur l'application testée, que ce soit une montée en charge soudaine du nombre d'utilisateurs qui se connecte en même temps ou à l'inverse une baisse de charge.

# TEST DE PIC DE CHARGE



## Douglas

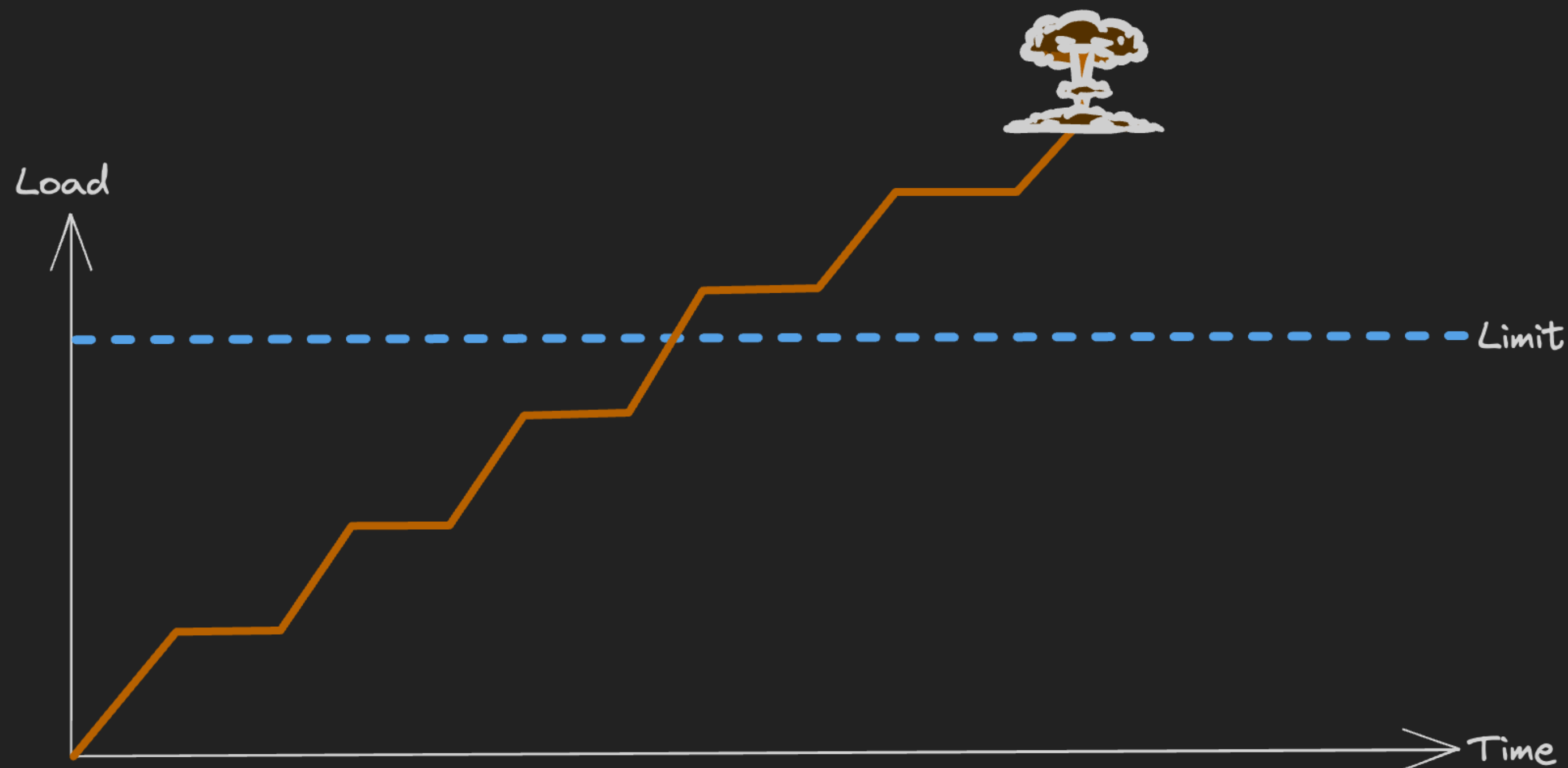
Le test aux limites permet d'évaluer le point de rupture de l'application. On parle souvent de test de capacité, car il est utile pour déterminer si le SLA est judicieux pour l'application testée.

L'objectif visé est la rupture de l'application pour une infrastructure donnée.

La rupture étant caractérisée par des données définies en accord avec le client par ex : "au doigt mouillé la limite sera de 70% des réponses KO", ou bien un temps de réponse de 10s par page...

La montée en charge est ici lente et le test peut être long.

# TEST AUX LIMITES



# MICROBENCHMARK

Outils utilisé : Open JDK JMH

Speaker notes

Pour illustrer ce test, je vous propose de comparer la recherche d'un objet dans une liste en comparant deux méthodes différentes :

La boucle célèbre FOR

vs

L'illustre Stream API de Java

Pour cela, nous utiliserons JMH.

C'est un outil proposé et maintenu par OpenJDK

# CONCEPTS

Type d'analyse

Définition d'un State

Écriture du code à tester



- Throughput : pour un temps donné, compte le nombre d'exec. de la fonction
- AverageTime : pour un nombre d'exécutions donné, mesure le temps
- SampleTime : exécute la fonction en continu et échantillon de temps
- SingleShotTime : exécute une fois la fonction, idéal pour exec à froid

## LES TYPES D'ANALYSES

- Throughput: ops/time
- AverageTime: time/op
- SampleTime: sampling exec time
- SingleShotTime: exec once

# STATE

```
1 @State(Scope.Benchmark)
2 public static class ExecutionPlan {
3     public List<Brand> brandList = new ArrayList<>(10000);
4
5     @Setup
6     public void setup() throws IOException {
7         for (int i = 0; i < 10000; i++) {
8             Brand builtBrand = buildBrand();
9             brandList.add(builtBrand);
10        }
11    }
12 }
```

## Speaker notes

- Une liste contenant un nombre certain de valeurs différentes
- Une fonction pour remplir notre liste. *Remarque* : la fonction 'buildBrand()' n'importe pas ici

- Une fonction qui utilise une 'stream' pour chercher par ID dans la 'List'
- Une fonction qui utilise une boucle 'ForEach' pour chercher par ID dans la 'List'

## CODE À TESTER

```
1 private Brand findBrandStream(List<Brand> brandList, Integer id) {
2     return brandList.stream().filter(brand -> brand.id().equals(id))
3         .findFirst()
4         .orElse(null);
5 }
6
7 private Brand findBrandFor(List<Brand> brandList, Integer id) {
8     for (Brand brand : brandList) {
9         if (brand.id().equals(id)) {
10             return brand;
11         }
12     }
13     return null;
14 }
```

Fonction de recherche par Stream, deux paramètres :

- Blackhole (pour éviter le "dead code eviction" du compilateur)
- Plan (Context)

Une boucle itérant un certain nombre de fois sur le code à analyser. On passe au Framework se nombre pour ses calculs

Le résultat de la fonction est envoyé dans le Blackhole,

L'ID de l'objet à trouver est pris de façon aléatoire dans la liste

# DÉFINITION DU BENCHMARK 1/2

```
1 @Benchmark
2 @OperationsPerInvocation(10000)
3 public void findBrandStream(Blackhole blackhole, ExecutionPlan plan) {
4     for (int i = 0; i < 10000; i++) {
5         blackhole.consume(
6             findBrandStream(
7                 plan.brandList,
8                 plan.brandList.get(random.nextInt(10000)).id()
9             )
10        );
11    }
12 }
```



## DÉFINITION DU BENCHMARK 2/2

```
1 @Benchmark
2 @OperationsPerInvocation(10000)
3 public void findBrandFor(Blackhole blackhole, ExecutionPlan plan) {
4     for (int i = 0; i < 10000; i++) {
5         blackhole.consume(
6             findBrandFor(
7                 plan.brandList,
8                 plan.brandList.get(random.nextInt(10000)).id()
9             )
10        );
11    }
12 }
```

En Java le code est compilé, mais pas trop. Une phase importante pour les applications Java a lieu à l'exécution, l'optimisation. Celle-ci est effectuée par le JIT Compiler. Pour éviter de mesurer ce travail d'optimisation, le Framework effectue quelques tours de chauffe

Puis mesure réellement. Attention à éviter tout ce qui peut perturber le test si vous l'exécutez sur votre machine (l'agenda, les mails, les notifications Slack)

# EXÉCUTION

```
1 # Blackhole mode: compiler (auto-detected, use -Djmh.blackhole.autoDetect=f
2 # Warmup: 3 iterations, 10 s each
3 # Measurement: 3 iterations, 10 s each
4 # Timeout: 10 min per iteration
5 # Threads: 1 thread, will synchronize iterations
6 # Benchmark mode: Throughput, ops/time
7 # Benchmark: org.example.StreamVsForMain.findBrandFor
8
9 # Run progress: 0,00% complete, ETA 00:03:20
10 # Fork: 1 of 1
11 # Warmup Iteration    1: 76239,113 ops/s
12 # Warmup Iteration    2: 76767,384 ops/s
13 # Warmup Iteration    3: 76239,113 ops/s
14
15 Iteration    1: 76972,113 ops/s
16 Iteration    2: 76562,211 ops/s
17 Iteration    3: 76972,113 ops/s
```

Speaker notes

Une fois les mesures effectuées JMH vous sort un joli petit tableau de comparaison.

L'erreur que l'on voit ici, c'est "l'interval de confiance", c'est-à-dire l'écart de temps autour de la moyenne (score) dans lequel se trouve 99% des temps mesurés.

# RÉSULTATS

Benchmark	Mode	Cnt	Score	Error	Units
StreamVsForMain.findBrandFor	thrpt	3	76758,517 ±	559,708	ops/s
StreamVsForMain.findBrandStream	thrpt	3	43736,684 ±	1081,065	ops/s

Taurus est un kit de développement qui permet à la fois :

- De pouvoir exécuter des tests avec plusieurs frameworks différents de test (sans qu'il y ait une grande différence dans l'implémentation des scénarios de tests)
- De pouvoir exécuter des tests aussi bien en local que sur un cloud provider. (plateforme Blazemeter)

# TAURUS

The BlazeMeter logo consists of a red icon of three horizontal lines of increasing length, followed by the word "BlazeMeter" in a bold, dark blue sans-serif font.



Les tests de charge avec Taurus sont assez simple, ils sont constitué de deux parties :

- la configuration de l'exécution
- la configuration du/des scénario(s)

# TAURUS

## in local-dev condition

```
execution:  
- concurrency: 200  
  hold-for: 1h  
  ramp-up: 15m  
  scenario: scenario1  
  steps: 5  
  throughput: 4000
```

```
scenarios:  
  scenario1:  
    requests:  
    - body: [...]   
      headers: [...]   
      label: first_req  
      method: GET  
      url: https://api.publicapis.org/entries
```

Les tests de charge avec Taurus sont assez simple, ils sont constitué de deux parties :

- la configuration de l'exécution
- la configuration du/des scénario(s)

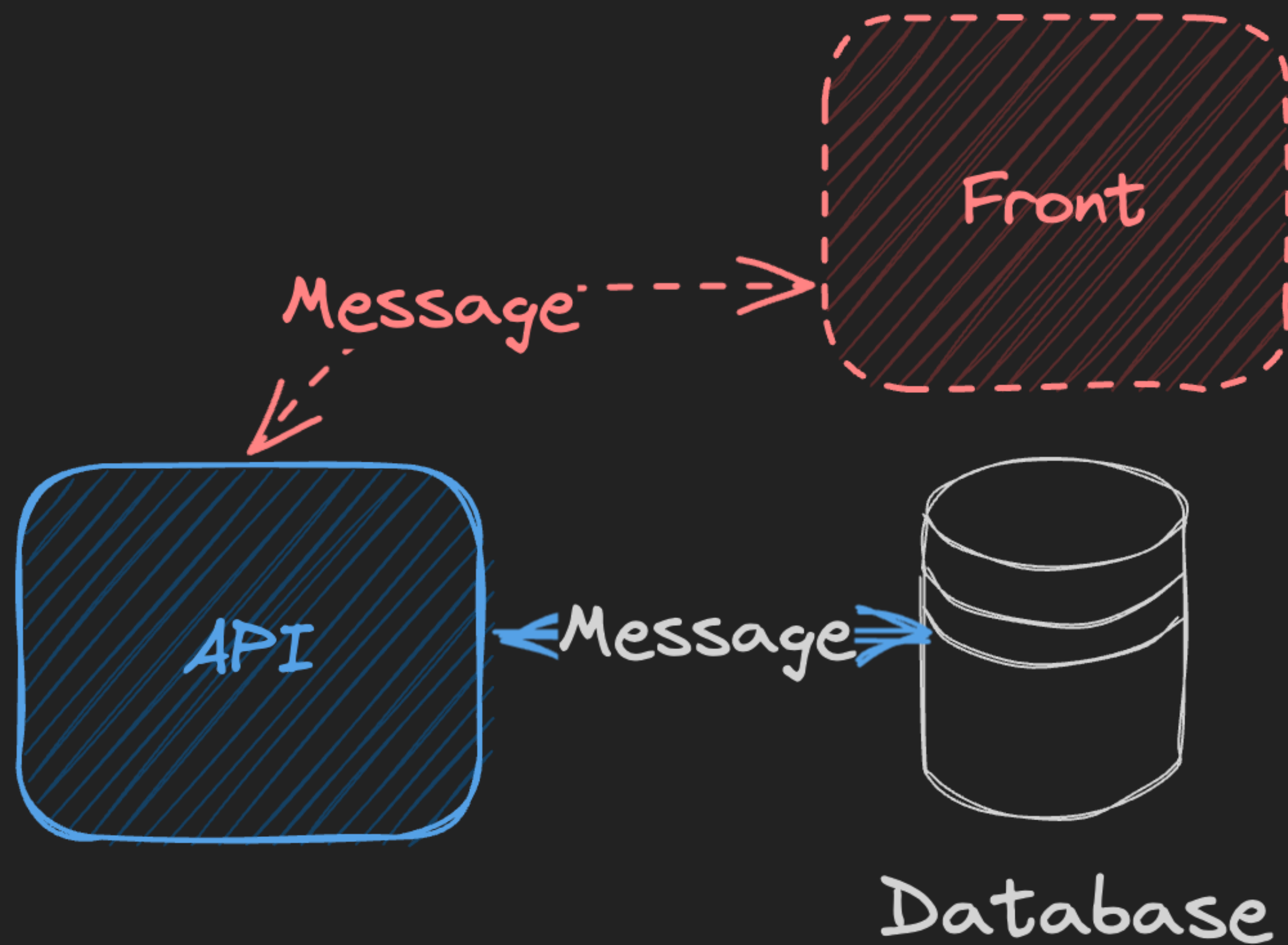
# TAURUS

## in cloud condition

```
execution:  
- concurrency: 200  
  hold-for: 1h  
  ramp-up: 15m  
  scenario: scenario1  
  steps: 5  
  throughput: 4000  
  provisioning: cloud  
  locations:  
    eu-central-1: 2  
    eu-west-1: 2
```

```
scenarios:  
  scenario1:  
    requests:  
    - body: [...]  
      headers: [...]  
      label: first_req  
      method: GET  
      url: https://api.publicapis.org/entries
```

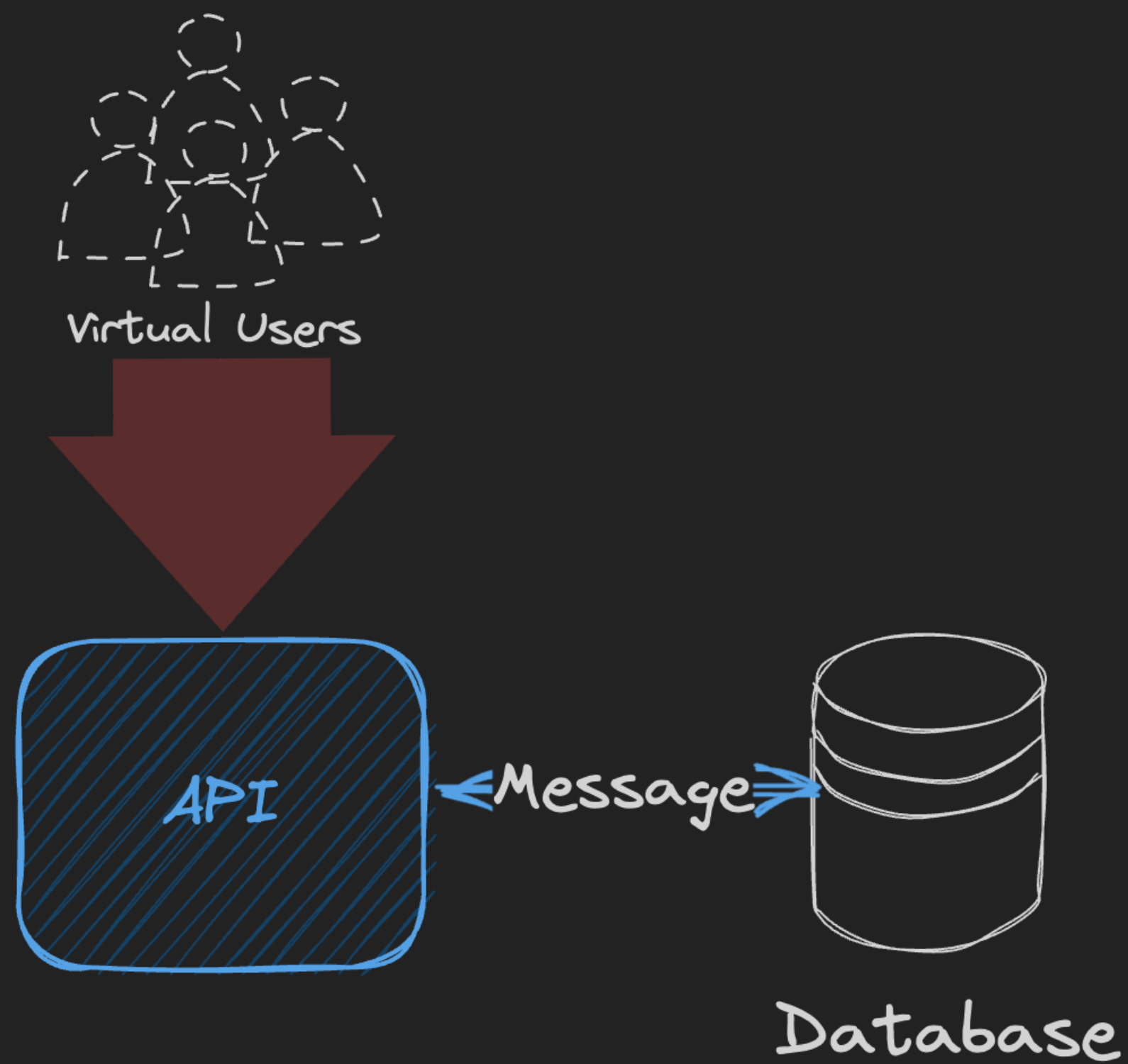
# TAURUS



Tout d'abord les paramètres d'exécution qui gère donc la volumétrie et le temps d'exécution du scénario de test:

- 10 utilisateurs
- 20 requêtes par secondes (total reparti sur les 10 utilisateurs)
- sur 20 minutes
- 10 secondes de montée en charge

# TAURUS



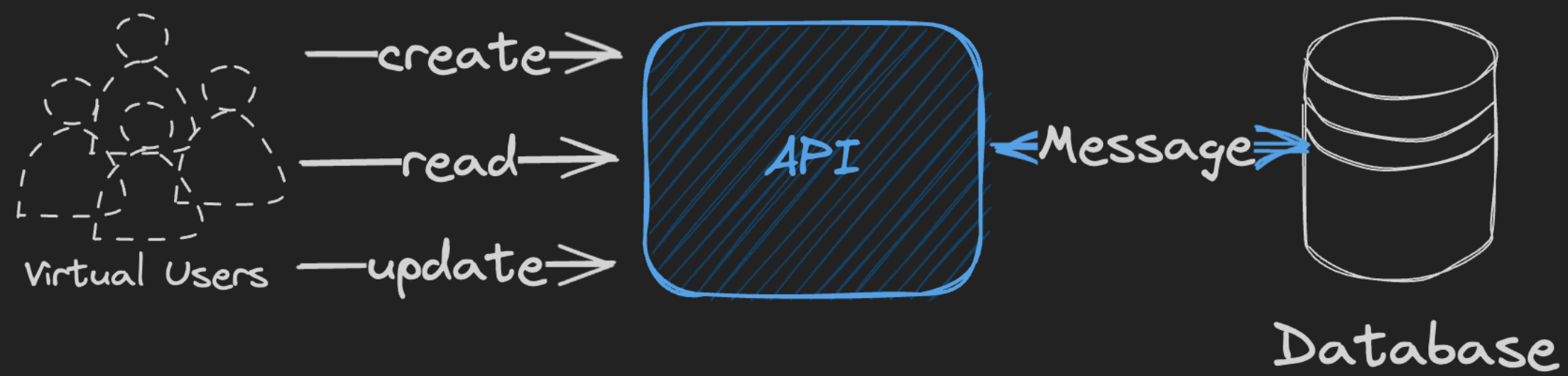


Pour tester notre API, on va donc avoir un scénario assez simple qui va simuler:

1. un enregistrement d'une donnée
2. une lecture de ce nouvel enregistrement
3. une modification de ce nouvel enregistrement

Ce scénario sera exécuté par tous les utilisateurs virtuels

# TAURUS



# TAURUS

## Demo

Speaker notes

**Dans cette demonstration, on fait :**

1. lancer le scenario en local
2. on regarde rapidement les stats si tout semble ok
3. lancer le scenario en cloud
4. on va regarder le scenario en mode cloud et consulter les différents tableaux de Blazemeter

# TAURUS

Local load test



Speaker notes

Lancement de Taurus en local

Voici les détails de ce qui est montré dans la vidéo :  
L'indicateur dans le coin supérieur gauche donne le nombre d'utilisateurs virtuel connecté. L'indicateur dans le centre gauche donne le nombre de requêtes par secondes. L'indicateur dans le coin inférieur gauche donne la latence moyenne des réponses. Le tableau dans le centre supérieur donne les statistiques des requêtes dans le dernier interval. Le tableau au centre donne les statistiques des requêtes globales. Les scénarios joués sont dans le coin supérieur droit. Les messages d'erreurs sont eux situé dans le coin inférieur droit.

# TAURUS

Speaker notes

**Les rapports a distance :**

- <https://a.blazemeter.com/app/#/accounts/1786851/work>  
Lancement de d'un test sur blazemeter (en mode cloud)

# QUESTIONS ?

# MERCI !

Présenté par :



Douglas SIX

SCAN ME



Sylvain  
LAVAZAIS