

**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

For this presentation, we're going to see roughly through the performance testing, and we're going to see how to work a solution to make performance testing easier and automated, "Taurus" by Blazemeter.

But first, let me introduce you what is performance testing and how many kinds of tests we can practice against an application.

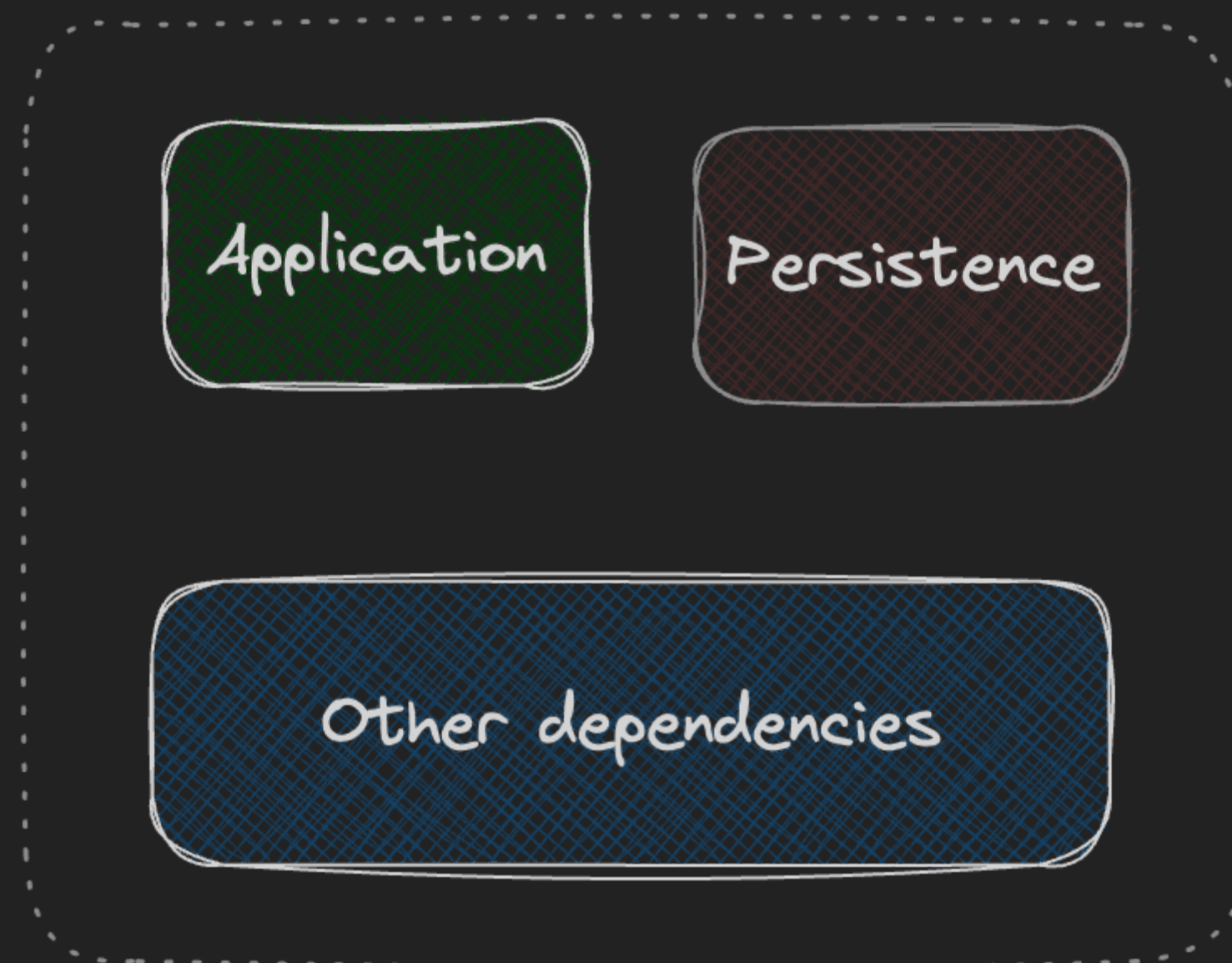
1. Les testes de performance, qu'est-ce que c'est ?
2. Quel type de test pratiquer?
  1. test en isolement
  2. test de charge
  3. test en stress
  4. test de pic
  5. test au limites
  6. test d'endurance
3. Tests en isolement avec Java Microbenchmark Harness
  1. Demo Isolement
4. Tests de charge 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. On peut ainsi en déterminer des axes d'amélioration pour l'application testée.

(point à discuter avec @sixdouglas)

# LES TESTES PERFORMANCE, 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 toute les réponses) et à un instant T.
- La quantité de ressources utilisées
- Mesure de la capacité d'adaptation / de la scalabilité

# LES TESTES PERFORMANCE, QU'EST-CE QUE C'EST ?

temps de réponse    les ressources utilisées    l'adaptation



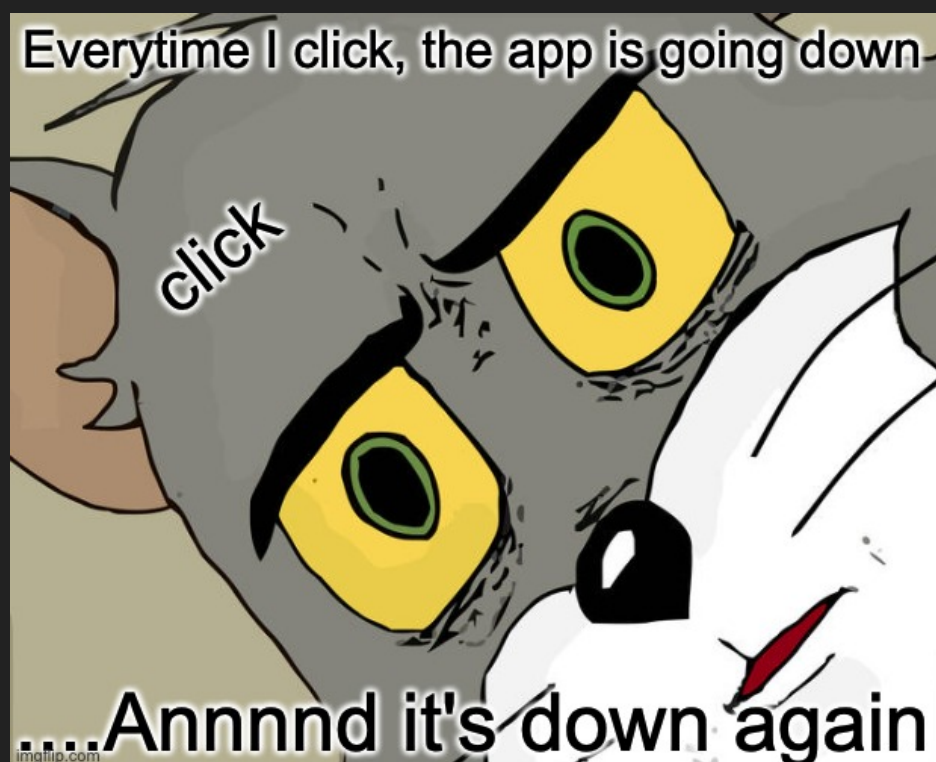


Les tests de performance permet également de :

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

# LES TESTES PERFORMANCE, QU'EST-CE QUE C'EST ?

les coupûres



bug / bottleneck



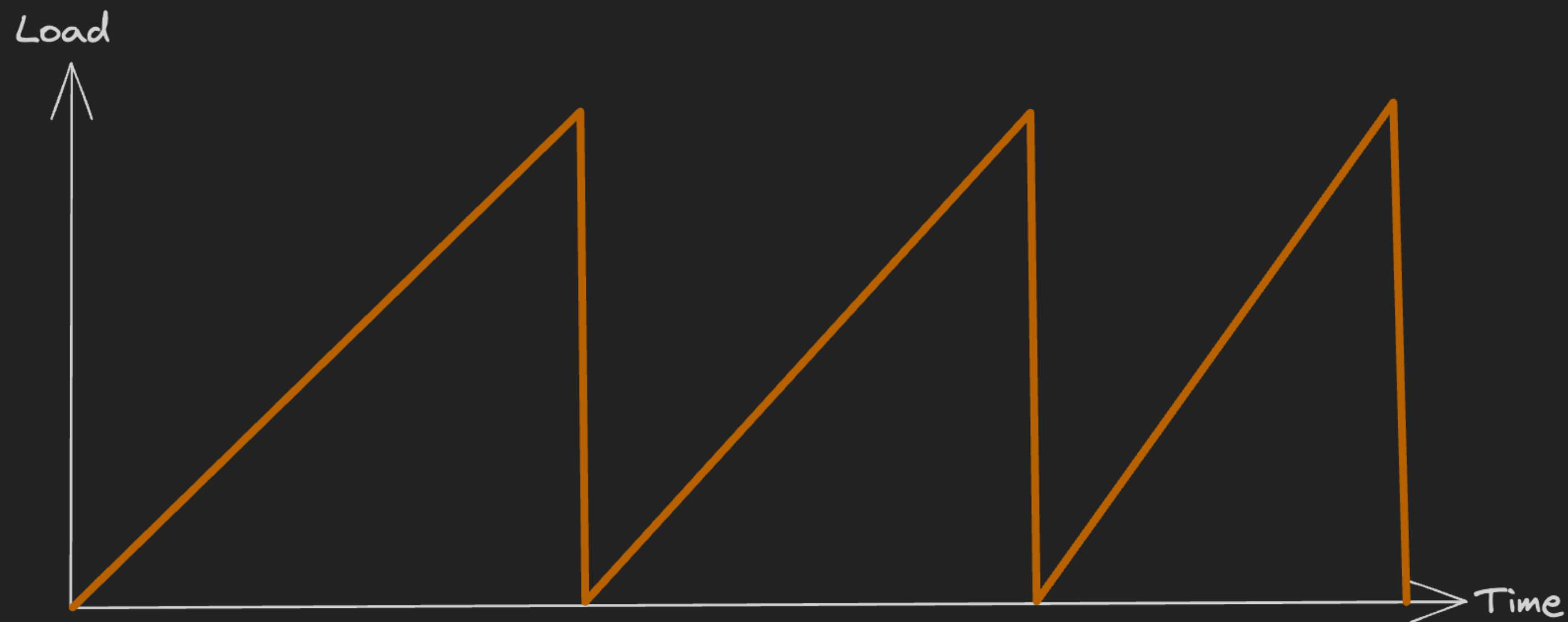
la sécurité



# QUEL TYPE DE TEST PRATIQUER?

1. test en isolement
2. test de charge
3. test en stress
4. test de pic
5. test aux limites
6. test d'endurance

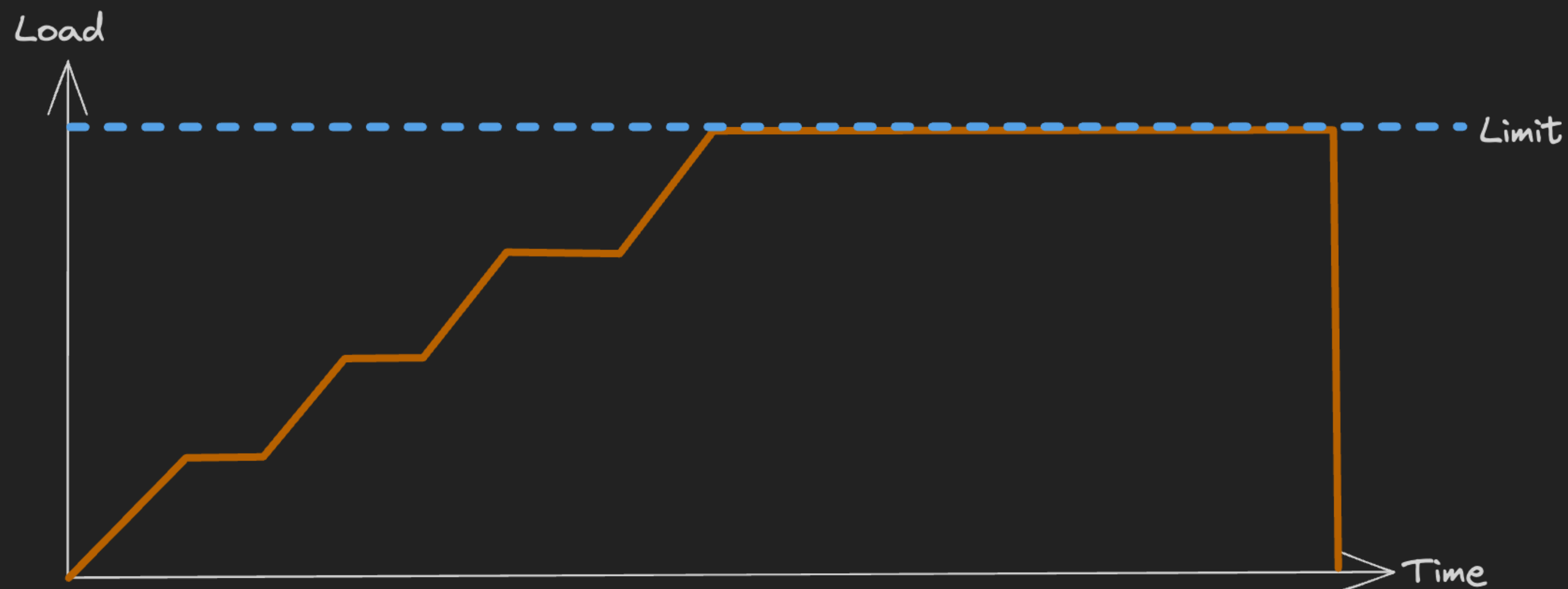
# TEST EN ISOLEMENT



Le test de charge est la forme la plus simple 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 décidées préalablement (par exemple au travers d'un SLA). L'infrastructure est également sous-monitoring durant cette phase de test.

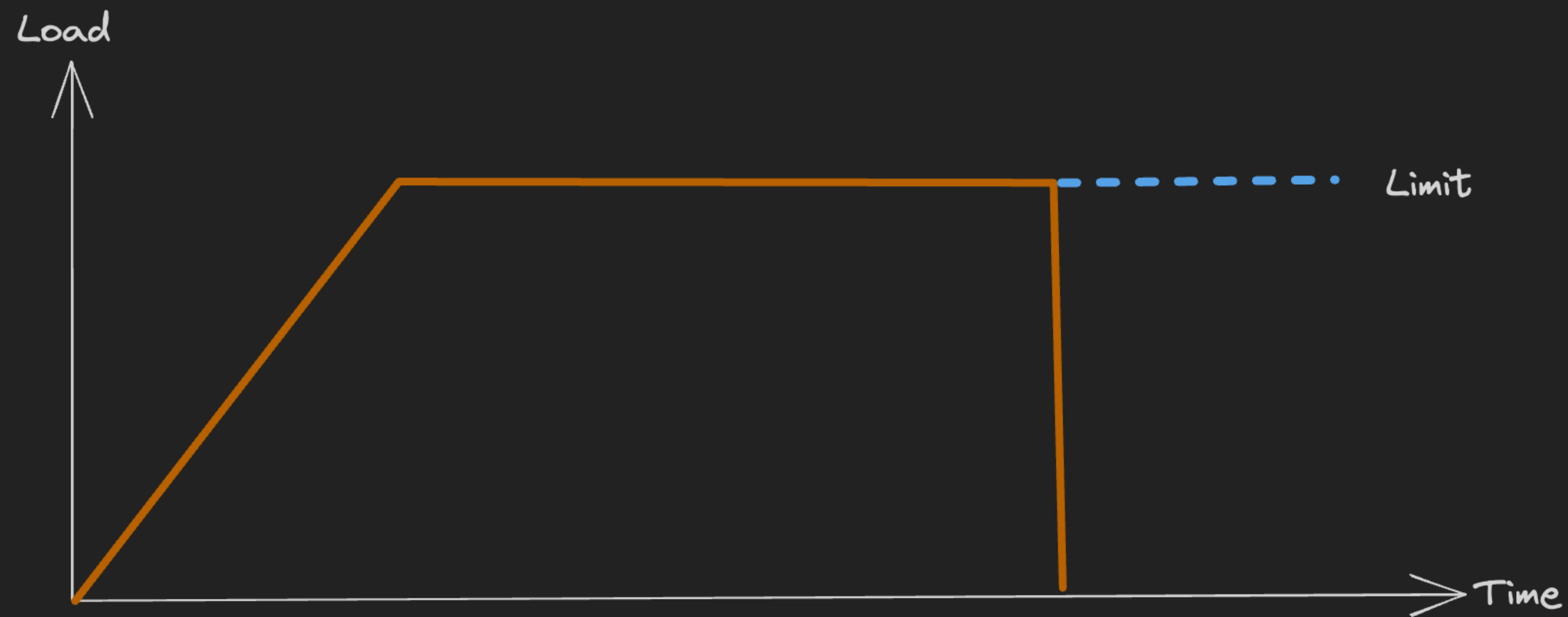
Ce type de test peut être utilisé comme un test de qualité d'une release à une autre, ainsi qu'un objectif à maintenir.

# TEST DE CHARGE



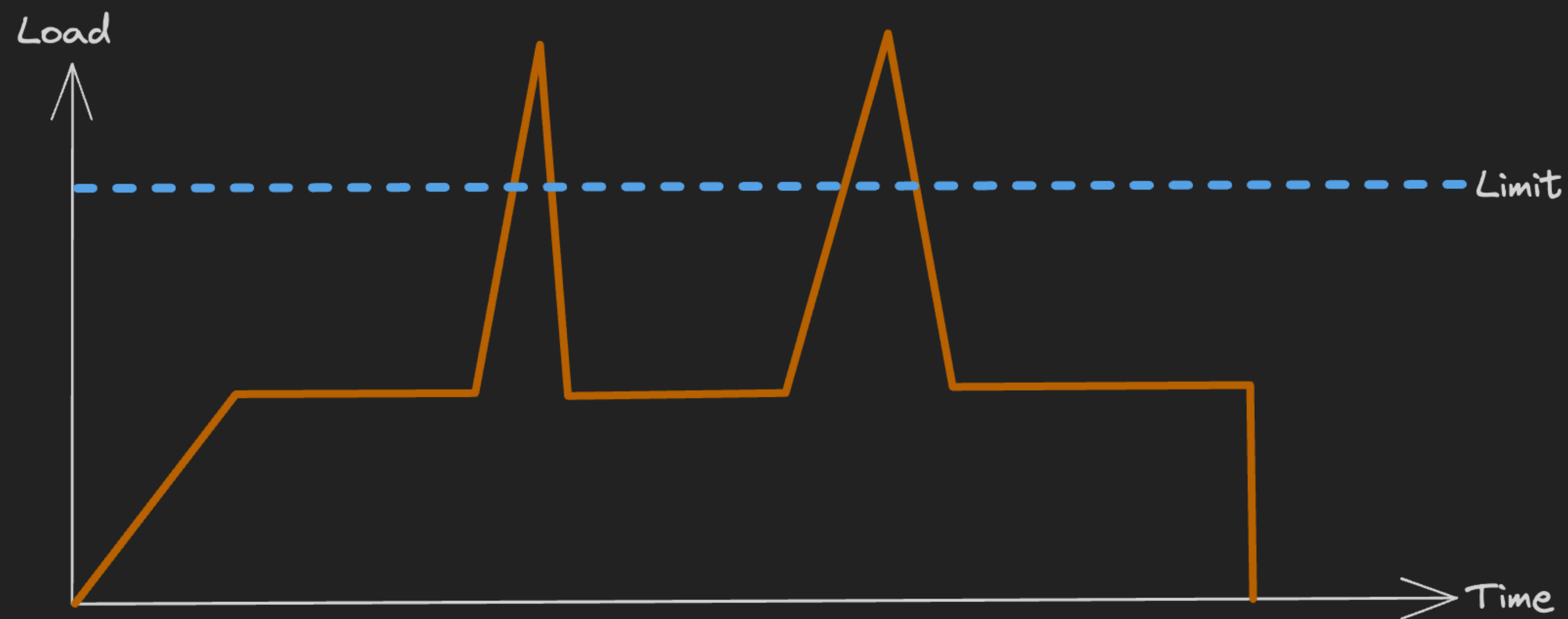


# TEST EN STRESS



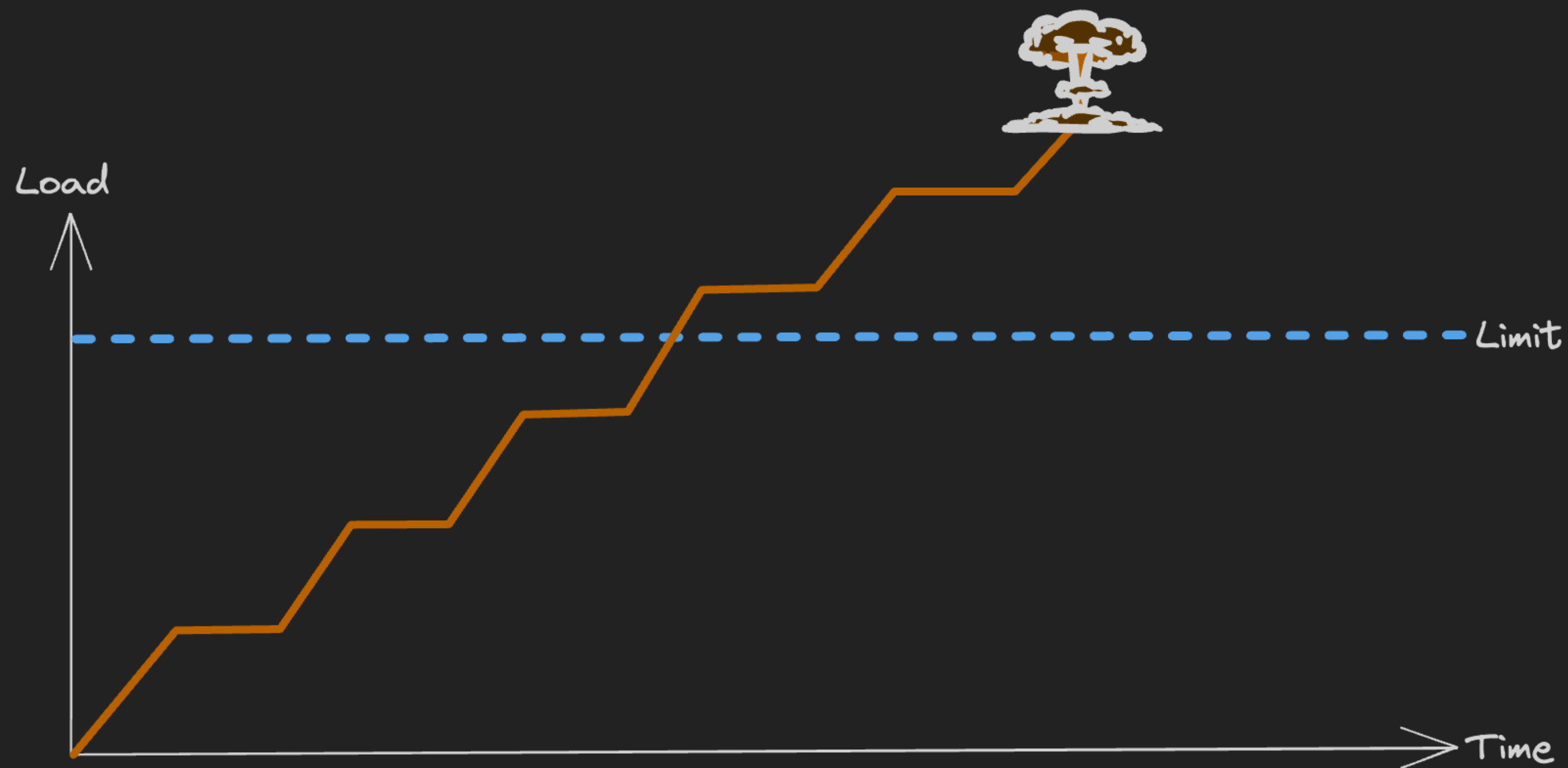
Le test de pic vise à déterminer les problèmes de performance quand un changement de contexte se produit sur l'application testée, que ce soit une montée en charge soudaine du nombre d'utilisateurs qui se connecte en même temps ou à l'inverse une baisse de charge.

# TEST DE PIC



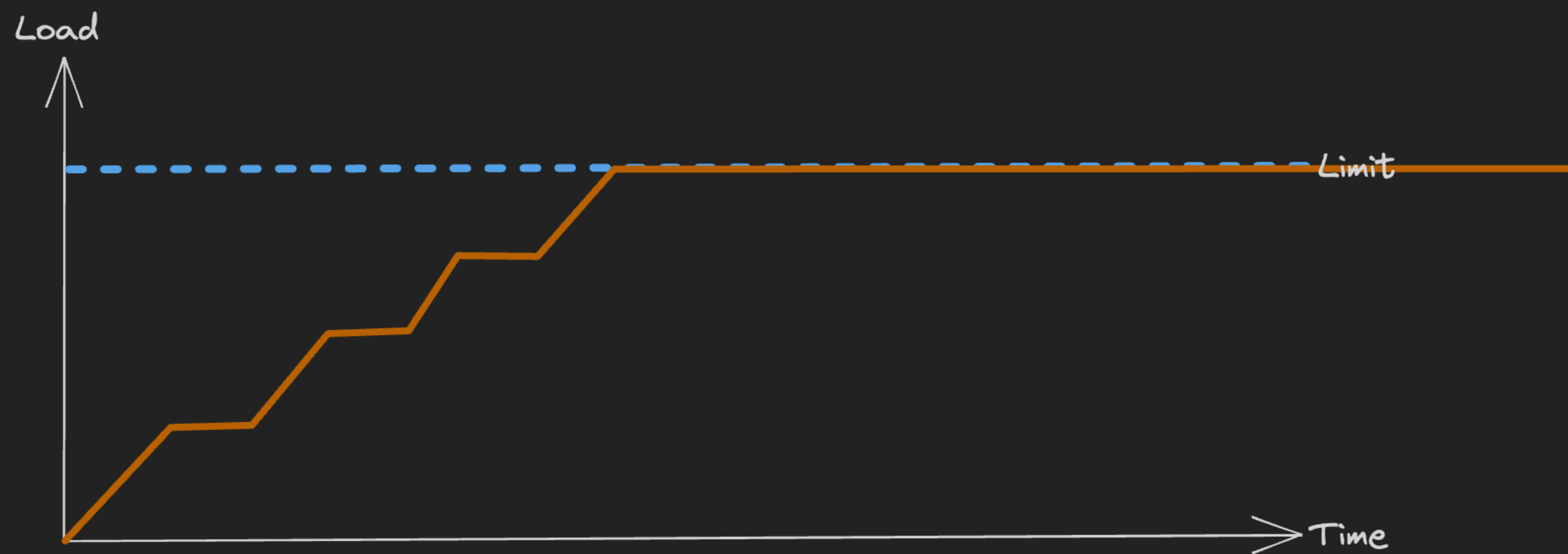
Le test au limites permet d'évaluer le point de rupture de l'application. On parle souvent de test de capacité car il est utile pour déterminer si le SLA est judicieux pour l'application testée.

# TEST AUX LIMITES



Le test d'endurance consiste à maintenir une charge pendant un temps extrêmement long et permet déterminer si l'application testée est capable de supporter un tel scénario. Comme les tests de charge l'infrastructure est également sous-monitoring durant cette phase de test.

# TEST D'ENDURANCE





# MICROBENCHMARK

Outils utilisé: **Open JDK JMH**

# CONCEPTS

Choix du type d'analyse

Définition d'un State

Écriture du code à tester

Écriture du Benchmark

## TYPES D'ANALYSES

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

### Speaker notes

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

- Petite annotation qui va bien
- Une petite liste dans laquelle nous allons chercher
- Une fonction pour remplir notre liste

# 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 }
```



- 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 }
```

# 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 }
```

# EXÉCUTION

```
# Blackhole mode: compiler (auto-detected, use -Djmh.blackhole.autoDetect=false)
# Warmup: 5 iterations, 10 s each
# Measurement: 5 iterations, 10 s each
# Timeout: 10 min per iteration
# Threads: 1 thread, will synchronize iterations
# Benchmark mode: Throughput, ops/time
# Benchmark: org.example.StreamVsForMain.findBrandFor

# Run progress: 0,00% complete, ETA 00:03:20
# Fork: 1 of 1
# Warmup Iteration   1: 76239,113 ops/s
# Warmup Iteration   2: 76767,384 ops/s

Iteration    1: 76972,113 ops/s
Iteration    2: 76562,211 ops/s
```



# RÉSULTATS

Benchmark	Mode	Cnt	Score	Error	Units
StreamVsForMain.findBrandFor	thrpt	5	76758,517 ±	559,708	ops/s
StreamVsForMain.findBrandStream	thrpt	5	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: 100  
  hold-for: 1h  
  ramp-up: 15m  
  scenario: scenario1  
  steps: 10  
  throughput: 2000
```

```
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

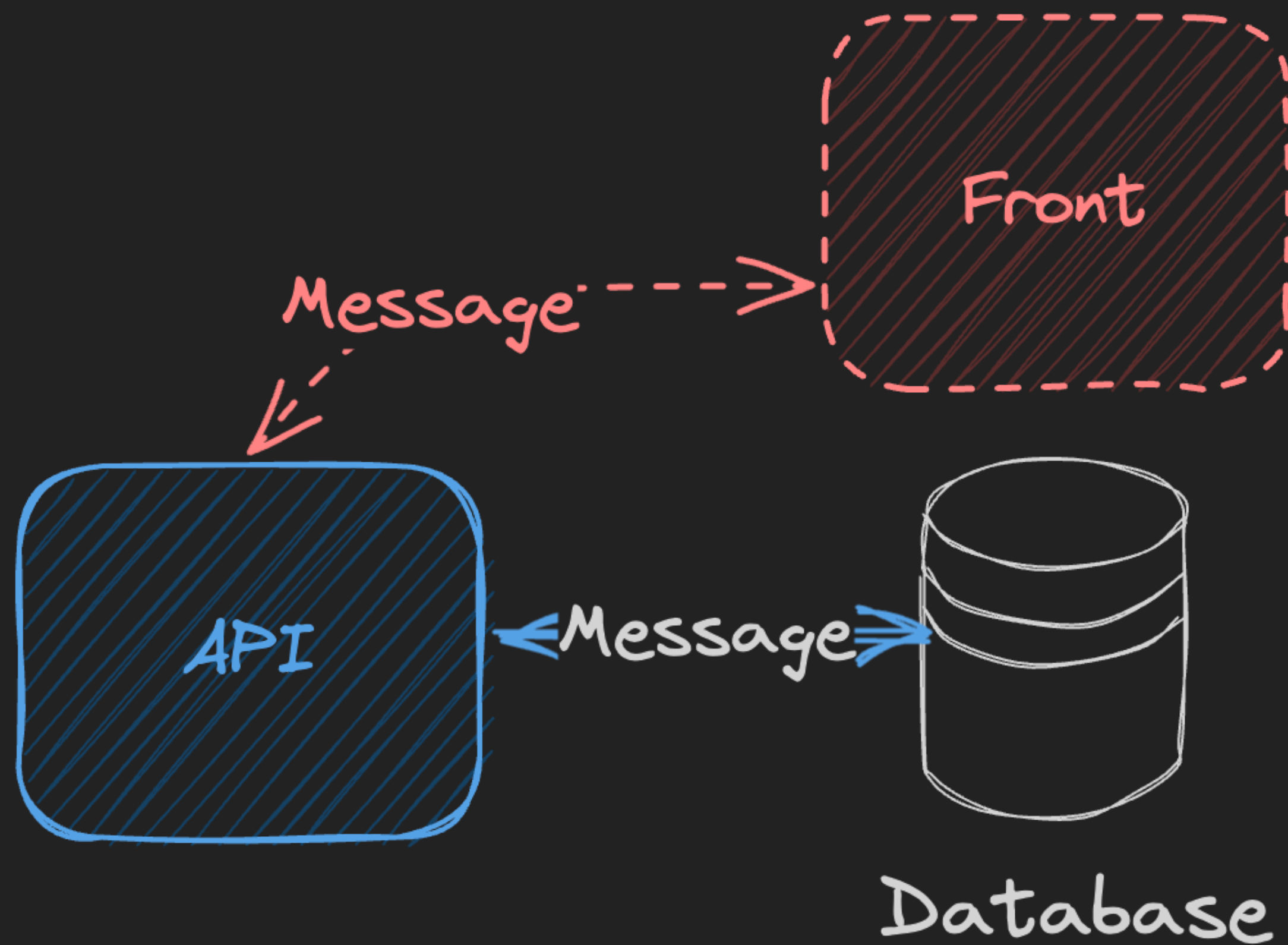
## in cloud condition

```
execution:  
- concurrency: 100  
  hold-for: 1h  
  ramp-up: 15m  
  scenario: scenario1  
  steps: 10  
  throughput: 2000  
  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
```



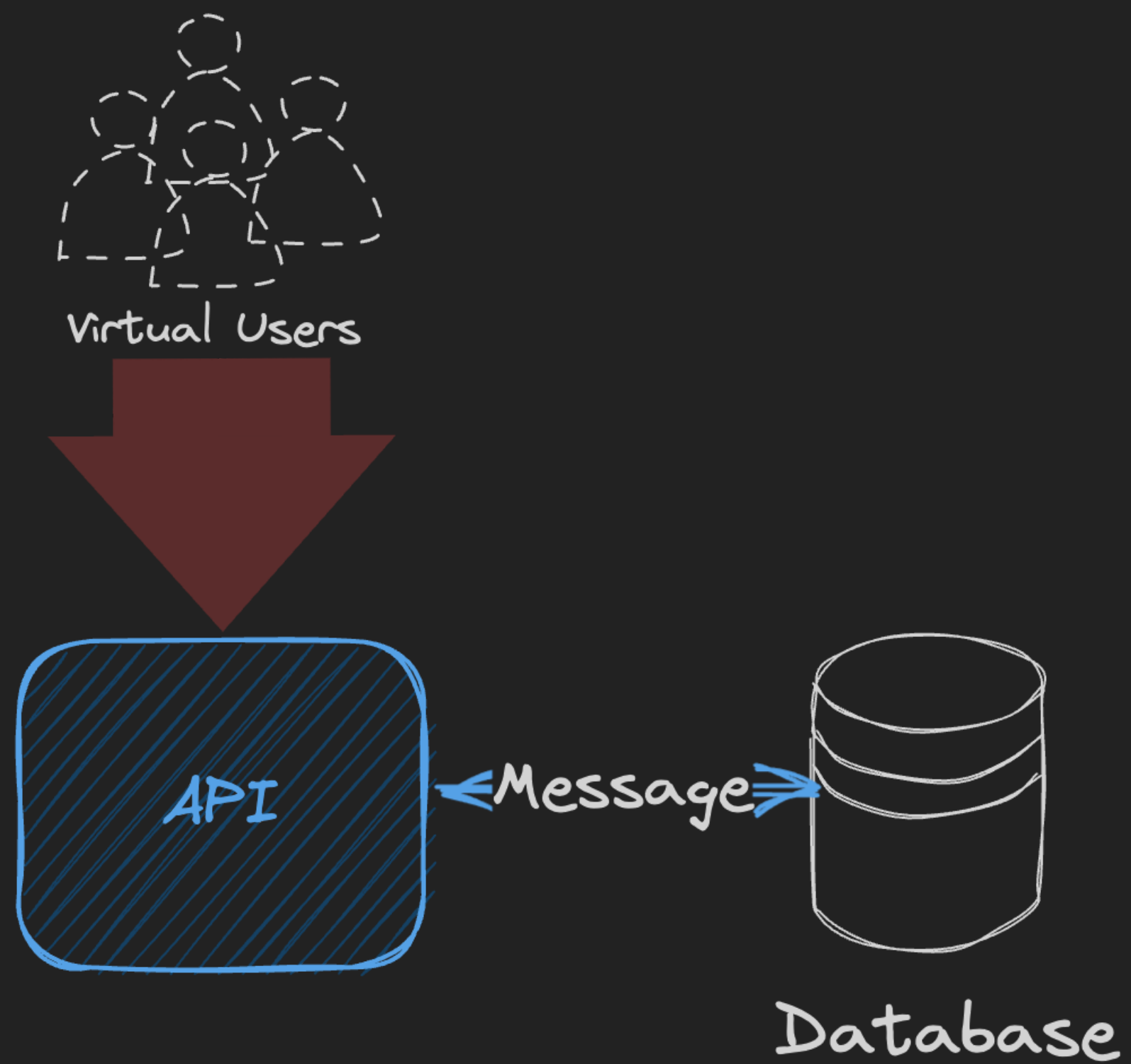
# TAURUS



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

- 100 utilisateurs
- 2000 requêtes par secondes (total repartit sur les 100 utilisateurs)
- sur 1 heure
- 15 minutes 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
4. suppression du message en question

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 tableau de Blazemeter

# MERCI !

Questions ?