EXTRAIT GRATUIT

Prometheus & Grafana

Le laboratoire d'observabilité

Apprendre le monitoring par des projets guidés : Docker, Python, Django, Linux

Un aperçu gratuit pour découvrir le style et la qualité du livre.

Cet extrait contient la Partie 1 — Fondations (en entier) et un aperçu du Projet 1.

Découvrir le portfolio en ligne :
https://grafanaportfolio.up.railway.app/

Ce que contient le livre complet

Un parcours complet, de la théorie jusqu'à un portfolio déployé en ligne, en trois parties et sept projets guidés.

Partie 1 — Fondations

- Comprendre le monitoring, l'observabilité et les métriques (les trois piliers, le modèle pull, les types de métriques).
- Installer le socle avec Docker, Prometheus et Grafana, et construire son premier tableau de bord.

Partie 2 — Sept projets guidés

- Surveiller une API Flask (métriques), puis y ajouter logs (Loki) et traces (Tempo).
- Surveiller une application Django avec django-prometheus.
- Monitorer un serveur Linux avec Node Exporter.
- Alertes et notifications : Alertmanager, email et Microsoft Teams.
- Monitorer une base MySQL avec mysqld_exporter.
- Surveiller une plateforme de vidéosurveillance (métriques métier).
- Mini NOC : la synthèse — application, base et serveur réunis sous une même supervision.

Partie 3 — Le Portfolio Django

- Construire une vitrine Django + Tailwind qui intègre en direct les tableaux de bord Grafana, et la déployer en ligne (Railway, Azure).

En plus

- Corrigés de tous les exercices et défis.
- Le code de tous les projets, prêt à l'emploi.

Partie 1 : Fondations

Avant de construire, il faut comprendre ce que l'on construit et pourquoi. Cette première partie pose les fondations : le vocabulaire du monitoring, le rôle de chaque outil, et l'installation du socle technique que tous les projets réutiliseront.

Figure 1 Illustration abstraite des fondations du monitoring et de l'observabilité.

Chapitre 1 : Comprendre le monitoring, l'observabilité et les métriques

Ce chapitre est le plus théorique du livre, et c'est volontaire. Les idées que vous allez rencontrer ici reviendront dans chaque projet. Une fois ce vocabulaire en place, tout le reste devient beaucoup plus simple. Prenez votre temps, mais ne vous inquiétez pas : dès le chapitre suivant, vous aurez les mains dans le cambouis.

> **MISSION**
>
> Imaginez que vous venez d'être embauche dans une petite entreprise qui édite une application web utilisée par ses clients. Tout fonctionne, jusqu'au jour où un client appelle, furieux : le site est lent, parfois il ne répond plus du tout.
>
> Le directeur informatique vous pose trois questions simples. Combien de requêtes l'application reçoit-elle ? En combien de temps répond-elle ? Combien d'erreurs renvoie-t-elle ? Vous ouvrez les serveurs, et vous réalisez que personne ne le sait. Il n'y a aucun instrument de mesure.
>
> Votre mission, tout au long de ce livre, sera de construire ces instruments. Ce chapitre vous explique de quoi ils sont faits.

1.1 Piloter sans tableau de bord

Conduire une application sans monitoring, c'est comme conduire une voiture dont le tableau de bord serait entièrement noir. Pas de compteur de vitesse, pas de jauge d'essence, pas de témoin de température. La voiture roule, tant qu'elle roule. Mais le jour ou un problème arrive, vous l'apprenez de la pire des manières : la voiture s'arrête au milieu de l'autoroute.

Un pilote d'avion, lui, dispose de centaines d'instruments. Ils ne rendent pas l'avion plus rapide. Ils ne corrigent pas les pannes tout seuls. Mais ils transforment l'inconnu en information. Le pilote sait, à chaque instant, dans quel état se trouve son appareil, et il peut agir avant que la situation ne devienne critique.

Le monitoring joue exactement ce rôle pour les logiciels et les serveurs. Il ne remplace pas l'ingénieur. Il lui donne les yeux pour voir. Et comme un bon tableau de bord, il doit être clair, fiable, et disponible au moment où l'on en a besoin, c'est-à-dire souvent au pire moment.

> **NOTE**
>
> Retenez cette différence essentielle. Le monitoring ne prévient pas les pannes par magie. Il rend visible ce qui était invisible, suffisamment tôt pour que vous puissiez réagir.

1.2 Monitoring et observabilité : deux mots, deux idées

On entend souvent les mots *monitoring* et *observabilité* employés l'un pour l'autre. Ils sont lies, mais ils ne disent pas la même chose, et la nuance est importante.

Le monitoring répond à des questions que vous avez prévues à l'avance. Vous savez que le processeur peut chauffer, alors vous surveillez le processeur. Vous savez que le disque peut se remplir, alors vous surveillez le disque. Le monitoring est l'art de poser des questions connues et de surveiller leurs réponses en continu.

L'observabilité va plus loin. C'est la capacite d'un système à vous laisser comprendre son état interne à partir de ce qu'il expose vers l'extérieur, y compris pour des questions que vous n'aviez pas anticipées. Un système observable vous permet d'enquêter sur un problème nouveau, jamais vu, sans avoir à rajouter du code et à tout redéployer.

Une façon simple de retenir la différence : le monitoring vous dit que quelque chose ne va pas. L'observabilité vous aide à comprendre pourquoi. Dans la pratique, on construit l'observabilité, et le monitoring en est l'usage quotidien.

Les trois piliers de l'observabilité

L'observabilité repose traditionnellement sur trois types de données, que l'on appelle les trois piliers. Chacun raconte une partie de l'histoire.

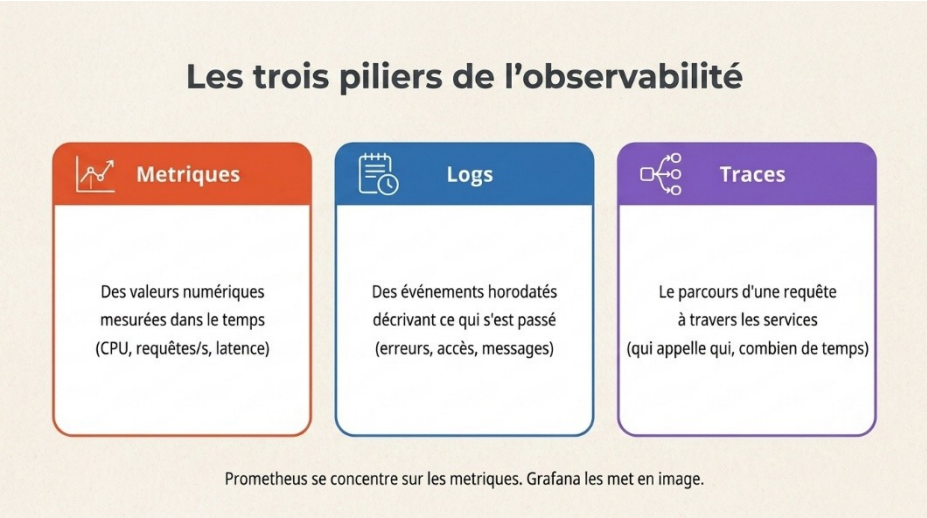

Figure 1.1 Les trois piliers de l'observabilité : métriques, logs et traces.

- **Les métriques** sont des nombres mesurés régulièrement dans le temps : le pourcentage de processeur utilise, le nombre de requêtes par seconde, la latence moyenne. Elles sont compactes, peu couteuses à stocker, et parfaites pour repérer une tendance ou déclencher une alerte.
- **Les logs** sont des messages horodates qui décrivent des évènements précis : une erreur, une connexion, un paiement refuse. Ils sont riches en détails mais volumineux.
- **Les traces** suivent le parcours d'une requête à travers plusieurs services et montrent ou le temps est passe. Elles sont précieuses dans les architectures distribuées.

Ce livre se concentre sur le premier pilier, les métriques, car c'est la spécialité de Prometheus. Grafana, de son coté, transforme ces métriques en graphiques lisibles. C'est par les métriques que l'on commence, parce qu'elles donnent rapidement une vision globale de la sante d'un système.

> **NOTE**
> Prometheus sait aussi déclencher des alertes à partir des métriques, et il s'intègre avec des outils de logs et de traces. Mais son cœur, sa raison d'être, ce sont les métriques. C'est pourquoi nous lui consacrons l'essentiel de notre attention.

1.3 Pourquoi Prometheus et Grafana

Il existe de nombreux outils de monitoring. Pourquoi ce livre choisit-il Prometheus et Grafana ? Pour trois raisons simples et solides.

La première est qu'ils sont gratuits et libres. Vous pouvez les installer chez vous, sur un serveur d'entreprise ou dans le cloud, sans payer de licence et sans limite artificielle. Vous apprenez sur les mêmes outils que ceux que vous utiliserez plus tard en production.

La deuxième est qu'ils sont devenus un standard de l'industrie. Prometheus est un projet de la ***Cloud Native Computing Foundation***, la même fondation qui héberge ***Kubernetes***. Savoir s'en servir est une compétence directement valorisable dans une carrière DevOps ou SRE.

La troisième est qu'ils forment un duo naturel. Prometheus est excellent pour collecter et stocker des métriques, mais son interface graphique est minimale. Grafana est excellent pour construire des tableaux de bord clairs et beaux, mais il ne collecte rien lui-même. Ensemble, ils se complètent parfaitement.

Qui fait quoi

- **Prometheus** collecte les métriques auprès des applications et des serveurs, les stocke dans sa propre base de données, et permet de les interroger avec un langage appelé PromQL. Il gère aussi les règles d'alerte.
- **Grafana** se connecte a Prometheus comme source de données, et affiche les métriques sous forme de graphiques, de jauges et de tableaux dans des tableaux de bord que vous concevez.
- **Les exporters** sont de petits programmes qui traduisent l'état d'un système (un serveur Linux, une base MySQL) en métriques que Prometheus sait lire.
- **Alertmanager** reçoit les alertes de Prometheus et les envoie au bon endroit : un courriel, un canal Teams, un message Slack.

1.4 Le modèle pull de Prometheus

Voici l'un des points les plus importants du chapitre, et l'un de ceux qui surprennent le plus les débutants. Avec Prometheus, ce ne sont pas les applications qui envoient leurs métriques au serveur. C'est l'inverse. C'est Prometheus qui va les chercher, a intervalle régulier. On appelle

cela le modèle **pull**, par opposition au modèle **push** ou ce serait l'application qui pousserait ses données.

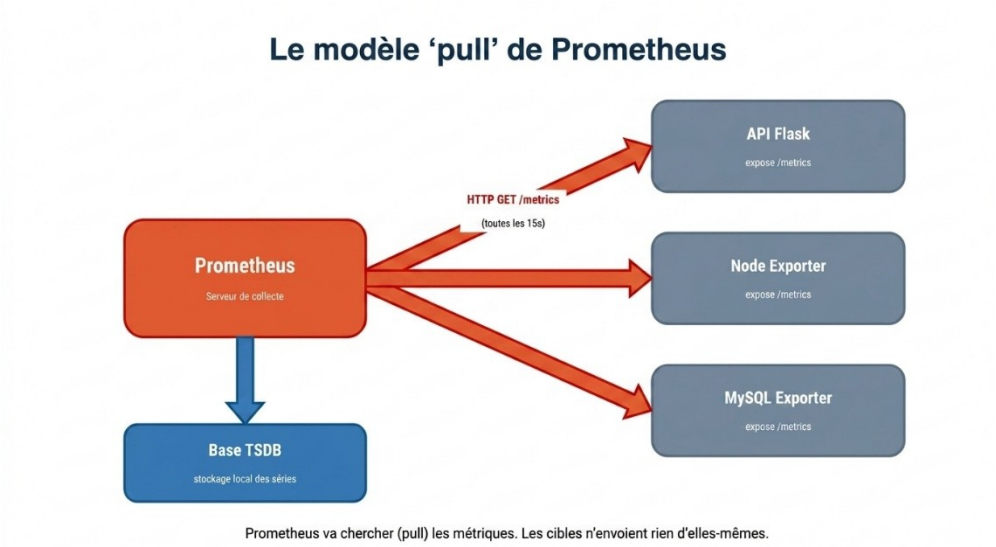

Figure 1.2 Prometheus va chercher (pull) les métriques auprès de chaque cible, à intervalle régulier.

Concrètement, chaque application ou serveur à surveiller expose ses métriques sur une adresse web, le plus souvent une page nommée **/metrics**. Prometheus connait la liste de ces adresses. Toutes les quinze secondes par exemple, il fait une simple requête **HTTP GET** sur chacune d'elles, lit la réponse, et enregistre les valeurs dans sa base. On appelle cette opération un **scrape**, que l'on peut traduire par collecte ou raclage.

Ce choix a des conséquences très pratiques. Comme c'est Prometheus qui déclenche la collecte, il sait immédiatement si une cible ne répond plus : la requête échoue. Il peut donc détecter qu'une application est tombée sans que celle-ci ait eu besoin de prévenir qui que ce soit. C'est élégant et robuste.

> **NOTE**
> Le vocabulaire à connaitre des maintenant :
>
> - **Cible (target).** Une adresse que Prometheus interroge, par exemple http://localhost:9090/metrics.
> - **Scrape.** L'action d'aller lire les métriques d'une cible.
> - **Job.** Un groupe de cibles qui jouent le même rôle, par exemple toutes les instances d'une même application.
> - **Endpoint /metrics.** La page web sur laquelle une application publie ses métriques.

A quoi ressemble une page /metrics ? A du texte tout simple, une ligne par mesure. En voici un extrait réaliste, celui que pourrait exposer une petite API.

```
# HELP http_requests_total Nombre total de requêtes HTTP recues.
```

```
# TYPE http_requests_total counter
http_requests_total{method="GET",code="200"} 1027
http_requests_total{method="POST",code="200"} 348
http_requests_total{method="GET",code="500"} 12

# HELP process_resident_memory_bytes Memoire utilisee par le processus.
# TYPE process_resident_memory_bytes gauge
process_resident_memory_bytes 5.24288e+07
```

Ne vous laissez pas impressionner. Chaque ligne utile est composée d'un nom de métrique, d'éventuelles étiquettes entre accolades, et d'une valeur. Les lignes commençant par un dièse sont des commentaires qui décrivent la métrique et son type. C'est tout. Voyons cela de plus près.

1.5 Anatomie d'une métrique

Une métrique Prometheus, ou plus précisément une série temporelle, se compose de trois éléments. Comprendre ces trois éléments, c'est comprendre quatre-vingts pour cent de Prometheus.

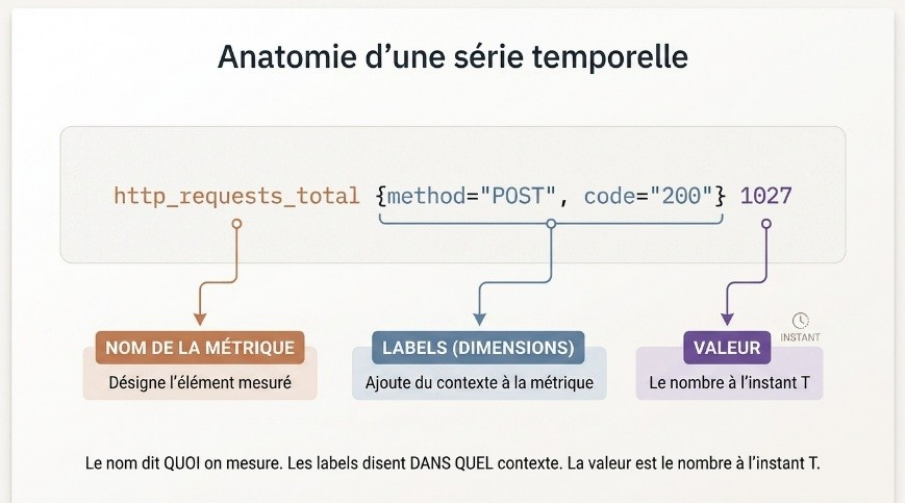

Figure 1.3 Une série temporelle : un nom, des étiquettes, une valeur.

1. **Le nom.** Il dit ce que l'on mesure. Par convention, il se lit de gauche à droite, du général au particulier, et se termine souvent par une unité : `http_requests_total`, `node_memory_used_bytes`, `http_request_duration_seconds`.
2. **Les étiquettes (labels).** Ce sont des paires clé-valeur entre accolades qui précisent le contexte. ***method="GET"*** et ***code="200"*** indiquent qu'il s'agit des requêtes GET ayant réussi. Les étiquettes sont ce qui rend Prometheus si puissant : une seule métrique peut se décliner en des milliers de séries selon les combinaisons d'étiquettes.
3. **La valeur.** Un nombre, mesure a un instant précis. Prometheus y ajoute automatiquement l'horodatage de la collecte. En empilant ces valeurs dans le temps, on obtient une série temporelle, c'est-à-dire une courbe.

Une analogie aide souvent. Pensez à une étagère de bibliothèque. Le nom de la métrique, c'est le titre de la collection, par exemple Romans policiers. Les étiquettes, ce sont les critères qui distinguent chaque livre : auteur, année, langue. La valeur, c'est le nombre d'exemplaires sur l'étagère à cet instant. Demain, ce nombre aura peut-être change, et c'est cette évolution qui nous intéresse.

> **ASTUCE**
> Bonne pratique de nommage : un nom de métrique décrit toujours une seule chose et indique son unité à la fin. Préférez `http_request_duration_seconds` à duration. Vos collègues, et vous-même dans six mois, vous remercieront.

1.6 Les quatre types de métriques

Toutes les métriques ne se comportent pas de la même façon. Prometheus distingue quatre types. Choisir le bon type au moment de créer une métrique est une décision importante, car elle détermine ce que vous pourrez en faire ensuite.

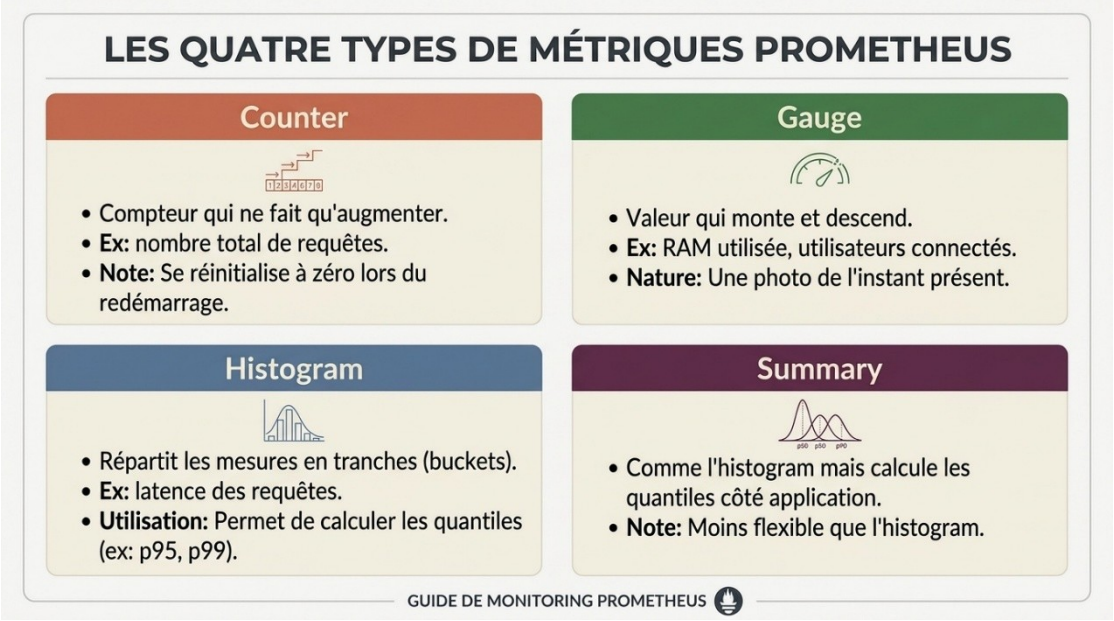

Figure 1.4 Les quatre types de métriques et leur usage typique.

Le Counter, ou compteur

Un **counter** est un nombre qui ne fait qu'augmenter. Il ne redescend jamais, sauf au redémarrage de l'application, ou il repart de zéro. On l'utilise pour compter des événements cumulatifs : le nombre total de requêtes reçues, le nombre total d'erreurs, le nombre total d'octets envoyés.

La valeur brute d'un counter n'est pas très parlante. Savoir que l'application a reçu 1 027 412 requêtes depuis son démarrage il y a trois semaines ne vous aide pas beaucoup. Ce qui vous intéresse, c'est le rythme : combien de requêtes par seconde en ce moment ? Nous verrons dans un instant que PromQL calcule cela pour vous.

La Gauge, ou jauge

Une gauge est une valeur qui peut monter et descendre librement. C'est une photo de l'instant présent. La température d'un processeur, la quantité de mémoire utilisée, le nombre d'utilisateurs connectes à cet instant, la place libre sur un disque : tout cela se mesure avec des gauges.

La différence avec un counter est facile à retenir. Si la valeur peut diminuer, c'est une gauge. Si elle ne peut que croitre, c'est un counter.

L'Histogram, ou histogramme

Un histogram repartit des mesures dans des tranches, appelées **buckets**. Il est conçu pour les valeurs dont on veut connaitre la distribution, comme la durée des requêtes. Plutôt que de retenir chaque durée individuelle, il compte combien de requêtes ont duré moins de 0,1 seconde, moins de 0,5 seconde, moins d'une seconde, et ainsi de suite.

Ce découpage permet ensuite de calculer des quantiles, par exemple le 95e centile de latence, souvent note p95. Le **p95** représente la durée en dessous de laquelle se trouvent quatre-vingt-quinze pour cent des requêtes. C'est une mesure bien plus honnête que la moyenne, car elle révèle l'expérience des utilisateurs les plus mal servis, ceux qui attendent le plus longtemps.

Le Summary, ou résume

Le ***summary*** ressemble à ***l'histogram*** : il sert aussi à observer des distributions et à calculer des quantiles. La différence est technique : le summary calcule les quantiles directement dans l'application, au moment de la mesure, tandis que l'histogram laisse Prometheus les calculer ensuite.

En pratique, l'histogramme est plus souple et plus souvent recommande, car il permet d'agréger les données de plusieurs instances et de choisir les quantiles après coup. Dans ce livre, nous privilégierons l'histogramme. Sachez simplement que le ***summary*** existe et a sa place dans certains cas précis.

> **ATTENTION**
> Erreur classique du débutant : utiliser un counter là où il faut une gauge, ou l'inverse. Posez-vous toujours la question : est-ce que ma valeur peut diminuer ? Si oui, c'est une gauge. Si elle ne peut que s'accumuler, c'est un counter.

1.7 Un premier regard sur PromQL

PromQL est le langage de requête de Prometheus. C'est lui qui transforme des séries temporelles brutes en informations utiles. Nous ne ferons ici qu'un premier survol ; chaque projet du livre approfondira PromQL au fil des besoins réels. L'objectif, pour l'instant, est seulement de ne pas être surpris quand vous le rencontrerez.

La requête la plus simple possible est le nom d'une métrique. Tapez le nom suivant, et Prometheus vous renvoie la dernière valeur connue de chaque série correspondante.

```
http_requests_total
```

Vous pouvez filtrer par étiquette pour ne garder que ce qui vous intéresse. Ici, seulement les requêtes en erreur.

```
http_requests_total{code="500"}
```

Rappelez-vous que la valeur brute d'un counter n'est pas parlante. La fonction rate calcule la vitesse moyenne d'augmentation par seconde, sur une fenêtre de temps que vous précisez entre crochets. La requête suivante donne le nombre de requêtes par seconde, en moyenne sur les cinq dernières minutes.

```
rate(http_requests_total[5m])
```

On peut aussi agréger. La fonction **sum** additionne plusieurs séries. Combinée a **rate**, elle donne le total de requêtes par seconde toutes routes confondues.

```
sum(rate(http_requests_total[5m]))
```

Ne cherchez pas à tout retenir maintenant. Retenez seulement trois idées. Un nom seul renvoie des valeurs instantanées. Les crochets, comme **[5m]**, demandent un intervalle de temps. Et des fonctions comme rate ou **sum** transforment ces données en quelque chose de lisible.

> **NOTE**
> Deux notions de vocabulaire PromQL qui reviendront souvent :
>
> - **Vecteur instantane.** Un ensemble de series avec une seule valeur chacune, celle du moment present. Exemple : `http_requests_total`.
> - **Vecteur de plage.** Un ensemble de series avec, pour chacune, toutes les valeurs sur une duree donnée. Exemple : `http_requests_total[5m]`. C'est ce dont **rate** à besoin pour travailler.

1.8 La chaine complète de monitoring

Mettons maintenant toutes les pièces ensemble. Voici à quoi ressemble une chaine de monitoring complète, celle que vous construirez progressivement tout au long des projets.

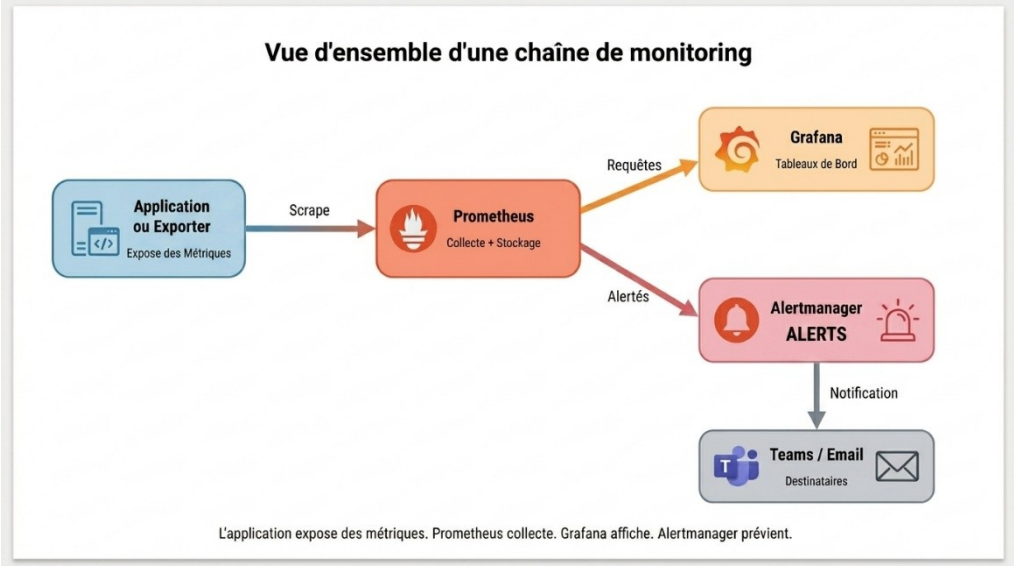

Figure 1.5 La chaine complète : de l'application surveillée jusqu'à la notification.

4. L'application, ou un exporter, expose ses métriques sur une page /metrics.
5. Prometheus va lire (**scrape**) cette page à intervalle régulier et stocke les valeurs dans sa base de séries temporelles.
6. Grafana interroge Prometheus avec PromQL et affiche les résultats dans des tableaux de bord.
7. En parallèle, Prometheus évalue des règles d'alerte. Quand une condition est remplie, il prévient **Alertmanager**.
8. Alertmanager met en forme l'alerte et l'envoie au bon destinataire : *courriel*, *Teams*, *Slack*.

Vous ne construirez pas tout cela d'un coup, et c'est tout l'intérêt de la démarche par projets. Le Chapitre 2 met en place Prometheus et Grafana. Le premier projet ajoute une application Flask à surveiller. Les projets suivants introduisent les ***exporters***, puis les alertes, puis une base de données, jusqu'à un véritable mini centre d'Operations. A chaque étape, vous ajoutez une pièce a un ensemble qui grandit.

1.9 Au-delà des métriques : Loki et Tempo

Prometheus et Grafana suffisent à construire un monitoring solide. Mais l'observabilité moderne va plus loin. Rappelez-vous les trois piliers du début de ce chapitre : les métriques ne sont que le premier. Cette section vous montre la cible vers laquelle ce livre vous emmené, pour que vous gardiez la vue d'ensemble à l'esprit dès le départ.

Au fil des projets, nous n'allons pas nous arrêter aux métriques. Nous ajouterons progressivement les deux autres piliers, les logs et les traces, jusqu'à obtenir une véritable plateforme d'observabilité. La bonne nouvelle est que chaque pilier dispose d'un outil de référence, et que ces trois outils sont conçus pour travailler ensemble au sein de Grafana.

Ces trois outils appartiennent au même écosystème, celui de **Grafana Labs**. Ils s'intègrent donc naturellement dans une seule et même interface : Grafana devient le poste de pilotage unique d'où vous regardez vos métriques, vos logs et vos traces sans changer d'outil.

Pilier	Outil de référence	Ce qu'il apporte
Métriques	Prometheus	Des nombres mesures dans le temps : charge, latence, taux d'erreurs.
Logs	Loki	Des messages horodatés qui racontent les événements précis : erreurs, accès, actions.
Traces	Tempo	Le parcours d'une requête à travers les services, et le temps passe à chaque étape.

Visuellement, l'architecture cible ressemble à ceci. Une même application alimente les trois piliers en parallèle. Chaque flux est collecté par son outil dédié, puis tout converge vers Grafana.

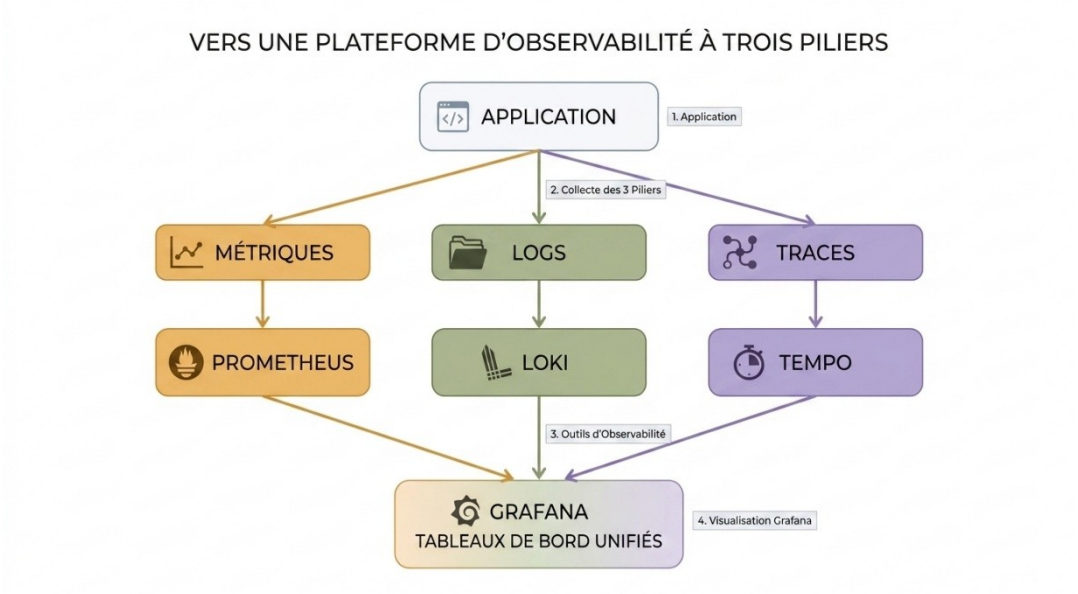

Figure 1.6 L'architecture cible : métriques, logs et traces réunis dans Grafana.

Ne soyez pas intimide par ce schéma. Vous ne le construirez pas d'un coup. Le Chapitre 2 met en place la colonne des métriques, avec Prometheus et Grafana. Les logs et les traces viendront se greffer projet après projet, une fois que les bases seront solides.

> **NOTE**
> Tous les projets n'ont pas besoin des trois piliers, et c'est une leçon importante. Sur un serveur Linux surveille par Node Exporter, par exemple, les traces n'apportent presque rien : il n'y a pas de parcours de requête à suivre. Vouloir les trois piliers partout, par principe, est une erreur de débutant.

> La bonne question n'est jamais combien de piliers puis-je ajouter, mais de quoi ai-je besoin pour répondre à mes questions. Chaque projet de ce livre vous aidera à développer ce jugement.

La feuille de route de l'observabilité

Une erreur fréquente, dans les cours et les livres, consiste à traiter d'abord toutes les métriques sur plusieurs projets, puis à ajouter les logs et les traces dans des chapitres isoles, tout à la fin. Le lecteur a alors l'impression de tout recommencer à zéro, et le lien avec les projets se perd.

Ce livre prend le parti inverse. Chaque pilier est introduit au moment où il devient utile, à l'intérieur d'un projet concret. Le tableau suivant montre quels piliers chaque projet mettra en œuvre. Vous le verrez se remplir au fil de votre progression.

Projet	Métriques	Logs	Traces
Projet 1 : API Flask	**Oui**	**Oui**	**Oui**
Projet 2 : Django	**Oui**	**Oui**	**Oui**
Projet 3 : Linux	**Oui**	**Oui**	Non
Projet 4 : AlertManager	**Oui**	**Oui**	Non
Projet 5 : MySQL	**Oui**	**Oui**	**Oui**
Projet 6 : Vidéosurveillance	**Oui**	**Oui**	**Oui**
Projet 7 : Mini NOC	**Oui**	**Oui**	**Oui**

> **ASTUCE**
> Gardez cette feuille de route sous les yeux. A la fin de chaque projet, vous pourrez cocher les piliers que vous maitrisez désormais. C'est un excellent moyen de mesurer votre progression et de rester motive.

Ainsi, chaque projet devient une étude de cas complète d'observabilité moderne, et vous apprenez les trois piliers naturellement, au rythme de vos besoins réels, plutôt que comme une théorie déconnectée de la pratique.

1.10 Idées reçues et points de vigilance

> **DEPANNAGE**
> Voici quelques malentendus fréquents, présentés sous forme de questions, pour éviter de partir sur de mauvaises bases.

Le monitoring ralentit-il mon application ? Très peu. Exposer des métriques coute quelques microsecondes par requête et une page web supplémentaire. C'est négligeable face au bénéfice.

Dois-je tout mesurer ? Non. Trop de métriques noient l'information utile et coutent cher à stocker. Commencez par les questions essentielles : combien de requêtes, combien de temps, combien d'erreurs.

Prometheus garde-t-il mes données pour toujours ? Non. Par défaut, il conserve les données pendant une durée limitée, quinze jours dans sa configuration de base. Pour le long terme, on utilise des solutions de stockage dédiées, hors du cadre de ce livre.

Une cible affichée DOWN signifie-t-elle toujours une panne ? Pas forcément. Cela signifie que Prometheus n'a pas réussi à lire la page /metrics. La cause peut être un mauvais port, une application arrêtée, ou un pare-feu. Nous verrons comment enquêter au Chapitre 2.

1.11 Exercices

Ces exercices ne demandent pas d'ordinateur. Ils vérifient que les idées du chapitre sont bien en place. Prenez un papier ou un fichier texte et répondez par vous-même avant de poursuivre.

1. Expliquez avec vos propres mots la différence entre monitoring et observabilité.
2. Pour chacune des grandeurs suivantes, dites si vous utiliseriez un counter ou une gauge : le nombre total d'erreurs depuis le démarrage, le nombre d'utilisateurs connectes en ce moment, la place libre sur le disque, le nombre total de connexions à la base de données.
3. Dans la ligne `node_filesystem_avail_bytes{device="/dev/sda1",mountpoint="/"} 12884901888`, identifiez le nom de la métrique, les étiquettes et la valeur.
4. Expliquez pourquoi la valeur brute d'un counter est rarement utile, et ce que la fonction rate apporte.
5. Décrivez, dans l'ordre, les cinq étapes de la chaine de monitoring de la Figure 1.5.

ASTUCE

Pour l'exercice 2 : souvenez-vous de la règle. Si la valeur peut diminuer, c'est une gauge. Les bonnes réponses sont donc counter, gauge, gauge, counter.

1.12 Défi de réflexion

Voici un défi plus ouvert, sans bonne réponse unique. Il vous prépare à penser comme un ingénieur SRE.

MISSION

Reprenez le scenario de la mission d'ouverture : une application web dont les clients se plaignent de lenteurs intermittentes.

Listez les cinq métriques que vous mettriez en place en priorité pour comprendre ce qui se passe. Pour chacune, indiquez son type (counter, gauge ou histogram) et la question à laquelle elle répond. Gardez votre liste : vous la comparerez avec ce que nous instrumenterons réellement dans le Projet 1.

1.13 Ce qu'on a appris

CE QU'ON A APPRIS

- Le monitoring rend visible l'état d'un système ; l'observabilité permet d'enquêter, y compris sur des problèmes imprévus.
- L'observabilité repose sur trois piliers : métriques, logs et traces. Prometheus se concentre sur les métriques.
- Prometheus utilise le modèle pull : il va lire (**scrape**) la page **/metrics** de chaque cible a intervalle régulier.
- Une série temporelle se compose d'un nom, d'étiquettes et d'une valeur horodatée.
- Il existe quatre types de métriques : counter, gauge, histogram et summary. Le choix du type est une décision importante.
- PromQL interroge les métriques. **rate** transforme un counter en vitesse, **sum** agrège plusieurs séries.
- La chaine complète relie l'application, Prometheus, Grafana et Alertmanager.

Vous avez maintenant le vocabulaire et les idées. Il est temps de passer à la pratique. Au prochain chapitre, vous installerez Prometheus et Grafana avec Docker, et vous verrez votre tout premier graphique.

PROJET 1

Surveillance de l'API Flask

Instrumenter, collecter et visualiser les métriques

Counter, **Gauge**, **Histogram**, PromQL et tableau de bord Grafana
Avec Python, Flask, Prometheus et Grafana

Prometheus & Grafana - Le laboratoire d'observabilité
Partie 2 : Projets

Projet 1 : Surveiller une API Flask

Votre premier projet réel. Vous allez écrire une petite API web en Python, lui apprendre à se décrire elle-même à travers des métriques, puis la surveiller avec Prometheus et Grafana. A la fin, vous aurez un tableau de bord vivant qui réagit à chaque requête, et vous comprendrez exactement ce qui se cache derrière chaque courbe.

Pourquoi commencer par une API Flask ? Parce que c'est l'exemple le plus parlant qui soit. Une API reçoit des requetés, met un certain temps à répondre, et échoue parfois. Ces trois réalités, le nombre de requetés, leur durée et leurs erreurs, sont précisément ce que tout ingénieur veut surveiller en premier. En les mesurant sur un cas simple, vous apprenez des gestes que vous réutiliserez sur n'importe quel service, quelle que soit sa taille.

Ce projet est aussi le moment ou les notions du Chapitre 1 prennent corps. Vous avez lu ce qu'étaient un counter, une **gauge** et un **histogram**. Vous allez maintenant en créer un de chaque, de vos propres mains, et voir immédiatement le résultat dans Prometheus puis dans Grafana. Le vocabulaire abstrait devient une expérience concrète.

Nous restons volontairement sur les métriques. Les **logs** et les **traces**, les deux autres piliers, viendront enrichir ce même projet plus tard, une fois ces fondations bien ancrées. C'est la promesse de la progression : ne jamais tout présenter d'un coup, et bâtir sur du solide.

3.1 Mission

> **MISSION**
>
> Une jeune entreprise expose une API a ses clients. L'API fonctionne, mais personne ne sait vraiment comment elle se comporte au quotidien. Quand un client signale une lenteur, l'équipe ne peut que hausser les épaules : aucune mesure, aucun historique, aucune preuve.
>
> Votre mission est de donner des yeux a cette équipe. A la fin du projet, le responsable doit pouvoir répondre, d'un coup d'œil sur un écran, a trois questions : ***combien de requetés l'API reçoit-elle en ce moment***, ***en combien de temps répond-elle***, et ***combien d'erreurs renvoie-t-elle***. Vous allez construire l'instrument qui répond à ces questions, en continu.

Gardez ces trois questions en tète tout au long du chapitre. Chaque métrique que nous créerons, chaque courbe que nous tracerons, existera pour répondre à l'une d'elles. C'est ainsi que l'on construit un monitoring utile : en partant des questions, pas des outils.

3.2 Architecture

Avant d'écrire la moindre ligne, visualisons l'ensemble. Le schéma suivant montre les quatre acteurs du projet et la façon dont ils communiquent.

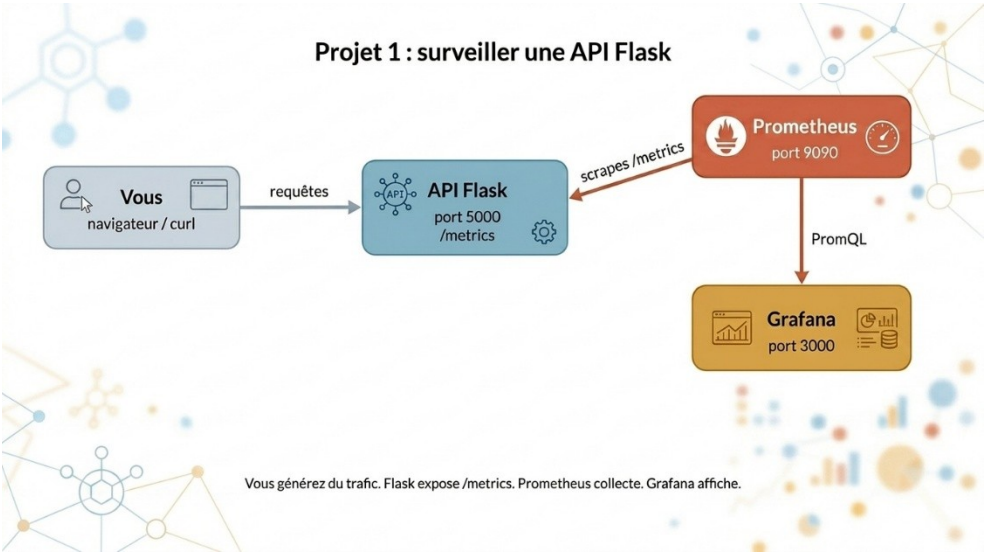

Figure 3.1 L'architecture du Projet 1 : vous gênerez du trafic, Flask expose ses métriques, Prometheus collecte, Grafana affiche.

1. Vous, depuis votre navigateur ou avec l'outil ***curl***, envoyez des requêtes à l'API Flask. C'est le trafic que nous allons mesurer.
2. L'API Flask traite chaque requête et, au passage, met à jour ses métriques. Elle les publie sur sa page ***/metrics***.
3. Prometheus va lire cette page toutes les cinq secondes et stocke les valeurs dans sa base.
4. Grafana interroge Prometheus avec PromQL et transforme ces valeurs en graphiques.

Vous reconnaissez la chaine du **Chapitre 1**. La seule nouveauté, mais elle est de taille, c'est que cette fois l'application surveillée, c'est vous qui l'écrivez. Vous décidez quelles métriques exposer. C'est ce qu'on appelle instrumenter une application.

3.3 Prérequis et structure du projet

Ce projet est autonome : il possède son propre dossier et son propre fichier docker-compose, indépendants du socle du **Chapitre 2**. Vous pourrez donc le lancer et l'arrêter sans toucher au reste. Avant de commencer, assurez-vous d'avoir les éléments suivants.

- Docker Desktop installe et démarre, comme au Chapitre 2.
- Visual Studio Code, ou l'éditeur de votre choix, pour écrire les fichiers.
- Votre terminal, Git Bash ou PowerShell, pour lancer les commandes.

Nous allons créer le dossier du projet et les fichiers suivants. Prenez un instant pour bien comprendre le rôle de chacun avant de les remplir.

```
Project1-Flask-Monitoring/
├── app.py              # l'API Flask et son instrumentation
├── requirements.txt    # les dépendances Python
├── Dockerfile          # comment construire l'image de l'API
├── docker-compose.yml  # les trois services : flask, prometheus, grafana
├── prometheus.yml      # ce que Prometheus doit collecter
└── alert.rules.yml     # nos premières règles d'alerte
```

> **NOTE**
> Vous trouverez ces six fichiers, prêts à l'emploi, dans le dossier ***Project1-Flask-Monitoring*** de votre laboratoire. Mais ne vous contentez pas de les copier. Saisissez-les vous-même, ou au moins lisez chaque explication : c'est en comprenant chaque ligne que vous apprendrez à instrumenter vos propres applications.

3.4 Construire l'API et l'instrumenter

Commençons par le cœur du projet, le fichier app.py. Nous allons le construire par morceaux pour bien comprendre chaque partie, puis vous verrez le fichier complet. Créez le fichier app.py dans le dossier du projet.

Les trois métriques

Tout en haut, après les imports, nous déclarons nos trois métriques. C'est ici que les types du Chapitre 1 deviennent du code. Lisez ce bloc lentement : chaque métrique correspond à l'une des trois questions de la mission.

```
from prometheus_client import Counter, Gauge, Histogram

# Counter : ne fait qu'augmenter. Combien de requêtes au total ?
REQUEST_COUNT = Counter(
    "http_requests_total",
    "Nombre total de requêtes HTTP recues.",
    ["method", "endpoint", "http_status"],
)

# Histogram : repartit les durées en tranches. En combien de temps ?
REQUEST_LATENCY = Histogram(
    "http_request_duration_seconds",
    "Duree des requêtes HTTP, en secondes.",
    ["method", "endpoint"],
)

# Gauge : monte et descend. Combien de requêtes en cours à cet instant ?
IN_PROGRESS = Gauge(
    "http_requests_in_progress",
    "Nombre de requêtes HTTP en cours de traitement.",
)
```

Observez le troisième argument de REQUEST_COUNT : la liste *method*, *endpoint*, *http_status*. Ce sont les labels, ces dimensions vues au Chapitre 1. Grace à eux, une seule métrique se décline en autant de séries qu'il y a de combinaisons : les GET sur ***/users*** qui réussissent, les POST en erreur, et ainsi de suite. C'est ce qui rendra nos graphiques riches.

> **ATTENTION**
> Choisissez vos labels avec soin. Un label dont les valeurs sont quasi infinies, comme un identifiant d'utilisateur ou un chemin exact contenant des numéros, crée une série différente à chaque valeur. C'est l'explosion de cardinalité, la première cause de saturation d'un Prometheus. Nous l'évitons plus bas en utilisant le motif de la route plutôt que le chemin exact.

Mesurer chaque requête

Comment mettre à jour ces métriques sans alourdir chaque route ? Flask offre deux points d'accroche bien pratiques : ***before_request***, ***execute*** avant chaque requête, et ***after_request***, ***execute*** après. Nous y démarrons un chronomètre, puis nous enregistrons la durée et incrémentons les compteurs.

```
import time
from flask import request

@app.before_request
def start_timer():
    request._start = time.time()
    if request.path != "/metrics":
        IN_PROGRESS.inc()

@app.after_request
def record_metrics(response):
    if request.path == "/metrics":
        return response
    elapsed = time.time() - getattr(request, "_start", time.time())
    # Le motif de la route, par exemple /users/<int:user_id>,
    # et non le chemin exact : on evite l'explosion de cardinalite.
    endpoint = request.url_rule.rule if request.url_rule else "unmatched"
    REQUEST_LATENCY.labels(request.method, endpoint).observe(elapsed)
    REQUEST_COUNT.labels(request.method, endpoint, response.status_code).inc()
    IN_PROGRESS.dec()
    return response
```

Trois détails méritent l'attention. D'abord, nous ignorons la page ***/metrics*** elle-même, pour ne pas polluer nos mesures avec les visites de Prometheus. Ensuite, nous utilisons *request.url_rule.rule*, c'est-à-dire le motif de la route comme ***/users/<int:user_id>***, et non le chemin réel comme ***/users/42***. Toutes les requêtes vers un utilisateur sont ainsi regroupées sous une seule série. Enfin, la méthode ***observe*** nourrit l'histogramme, tandis que ***inc*** et ***dec*** font respectivement monter le compteur et la gauge.

Vous voulez la suite ?

Cet extrait n'était qu'un aperçu. Le livre complet vous accompagne, pas à pas, de la première métrique jusqu'à un portfolio d'observabilité déployé en ligne.

En vous procurant le livre, vous obtenez :

- les trois parties et les sept projets guidés, complets, avec captures d'écran en couleur ;
- les corrigés de tous les exercices et défis ;
- une archive avec le code de tous les projets, envoyée par email après l'achat ;
- le portfolio complet (code corrigé) en référence.

Voir le portfolio en ligne, construit dans le livre :

https://grafanaportfolio.up.railway.app/

Merci de votre lecture — et bon monitoring !