

Se você me colar **um exemplo de log JSON real** (uma linha do console do ms-checkout-orchestrator), eu te devolvo:

- a query Loki exata (com as labels que o Promtail está usando no seu ambiente),
- e confirmo se o `trace_id` está aparecendo no lugar certo pra você clicar e abrir no Tempo.

eu preciso dos logs e tracer das aplicações rodando na minha maquina local, estou no linux ubuntu 24..... eu desenvolvo aplicações na maquina local..... e só depois que disponibilizo as imagens em container... mas antes disto.... toda a evolução, teste, observabilidade, monitoramento, logs, traces, alertas, alarmes precisam estar rodando junto comigo na minha maquina local.... para eu acompanhar pelos os dashboards do grafana... --->>> o codex esta me limitando: Logs no Loki só aparecem para serviços rodando em containers; se você roda pelo IntelliJ no host, precisa enviar logs para Loki (promtail no host ou appender em arquivo + volume).  
- Traces end-to-end dependem do OTEL SDK/exporter ativo; o dashboard de trace funciona com trace\_id (logs já têm esse campo).

Entendi perfeitamente — e **você não está pedindo nada impossível**. Só que aqui tem uma verdade técnica importante:

- **Traces** (Tempo) você já consegue coletar de apps rodando no host **sem container** (via **OpenTelemetry Java Agent** exportando OTLP → Tempo).

- **Logs** (Loki) o Grafana Loki **não “puxa” logs do host sozinho**. Ele precisa de um **“shipper”** (Promtail/Alloy/Fluent Bit) no host **ou** o app gravar logs em arquivo e alguém coletar, **ou** você mandar logs via protocolo próprio (menos comum). Isso não é limitação do Codex; é como Loki funciona.

A boa notícia: dá pra deixar tudo **rodando junto com você no Ubuntu**, enquanto você desenvolve no IntelliJ, com **uma experiência top**.

Vou te dar 2 arquiteturas (você escolhe depois; eu recomendaria a A).

## Arquitetura A (recomendada): Infra em Docker + Apps no Host (IntelliJ)

**Objetivo:** você roda os microserviços no IntelliJ; **Prometheus, Loki, Tempo, Grafana** rodam em Docker; e você vê tudo nos dashboards.

### Componentes

- **Métricas:** Spring Actuator/Micrometer → Prometheus (container) raspa o host
- **Traces:** OTel Java Agent no host → OTLP gRPC → Tempo (container)
- **Logs:** Logback JSON no host → **arquivo local** → Promtail (host) → Loki (container)
- **Dashboards:** Grafana (container)

✓ Isso é “modo dev profissional”.

### 1) Traces do host → Tempo (sem container)

Você já tem tempo com porta 4317. Então, em cada micro rodando no IntelliJ, você só precisa garantir o **javaagent** + env vars:

## IntelliJ VM Options (por micro)

Exemplo:

```
-javaagent:/caminho/opentelemetry-javaagent.jar
```

## Env vars (Run Configuration)

```
OTEL_SERVICE_NAME=ms-order
OTEL_EXPORTER_OTLP_ENDPOINT=http://localhost:4317
OTEL_EXPORTER_OTLP_PROTOCOL=grpc
OTEL_TRACES_EXPORTER=otlp
OTEL_METRICS_EXPORTER=none
OTEL_LOGS_EXPORTER=none
OTEL_RESOURCE_ATTRIBUTES=deployment.environment=local
```

**Importante no Linux:** como o app está no host, ele fala com localhost:4317 de boa (Tempo tá com 0.0.0.0:4317->4317).

Isso sozinho já te dá traces no Tempo para apps rodando no host.

## 2) Logs do host → Loki (o que faltava)

Aqui é onde o Codex te alertou, e está certo: **Loki não vê stdout do IntelliJ.**

A solução mais limpa é:

1. **logback** escrever também em arquivo **JSON** local
2. **promtail no host** ler esse arquivo e enviar pro Loki

## 2.1 Logback: gravar em arquivo (sem perder console)

Em cada micro, `logback-spring.xml` deve ter 2 appenders:

- Console JSON (pra você ver no IntelliJ)
- File JSON (pra promtail coletar)

Exemplo conceitual:

- arquivo em: `./logs/ms-order.json` (por micro)

Isso funciona perfeito.

## 2.2 Promtail rodando no host

Você instala e roda Promtail como processo/serviço no Ubuntu 24.

Ele vai “tail -f” dos arquivos `./logs/*.json` e mandar pro

Loki: `http://localhost:3100/loki/api/v1/push`

**Resultado:** logs dos apps rodando no IntelliJ aparecem no Loki **igual container**.

Dica de ouro: use label `service: ms-order` e env: `local` no promtail, assim suas queries ficam fáceis.

## 3) Métricas do host → Prometheus (seu caso)

Você já viu que Prometheus container não consegue raspar `host.docker.internal:809x` por causa de firewall/UFW e rede do compose.

O caminho mais robusto no Linux é **raspar o gateway real da bridge do compose** (ou liberar a subnet correta no UFW). Você já está no caminho.

A forma “padrão ouro”:

- no `prometheus.yml`, em vez de `host.docker.internal:8097`, usar o **IP do host no gateway da bridge** (ex.: `172.18.0.1:8097`)
- ou então liberar no UFW as subnets corretas ( `172.18.0.0/16` etc.)

Mas dá pra resolver sem gambiarra:

- **rodar o Prometheus no host também** (opção B abaixo) e aí ele raspa localhost sem stress.

## Arquitetura B (alternativa): Tudo no host (sem Docker pra observability)

Você instala no Ubuntu:

- Grafana
- Prometheus
- Loki
- Tempo
- Promtail

E deixa Docker só pra Kafka/Postgres etc.




✓ Prós: sem dor de rede, `localhost` funciona pra tudo

× Contras: mais instalação/máquina “suja”, upgrade mais chato

Eu só recomendo se você cansar da dor de rede do Docker → host (que dá pra resolver na A).

## Minha recomendação final (prática)

Fica com a Arquitetura A, mas completa o “triângulo”:

-  Tempo (container) recebendo traces do host via OTel agent
-  Promtail (host) enviando logs do host pro Loki (container)
-  Prometheus (container) raspando métricas do host (ajustando subnet UFW ou target IP)

Assim você desenvolve no IntelliJ e vê tudo no Grafana **em tempo real**, do jeito que você quer.

## O que eu quero de você pra te passar o passo-a-passo exato (sem chute)

Me diga só isto:

1. Você topa rodar **Promtail no host** (binário/serviço), ou prefere rodar Promtail também em Docker?
2. Onde você quer gravar os logs em arquivo? Ex.: `~/retail-store/logs/<service>.json`

Se você disser “sim, promtail no host e logs em `./logs/`”, eu já te passo:

- o `logback-spring.xml` definitivo (console+file)
- o `promtail.yaml` definitivo (com labels `service` automático)
- e o comando `systemd` pra subir promtail sempre junto com sua máquina.

rodar no docker... pois nao posso ficar intando e atualizando um monte de coisa na minha maquina

Pensou por ...

Perfeito — então vamos fazer **tudo “local + IntelliJ no host”**, mas **sem instalar nada no Ubuntu: toda a observabilidade continua em Docker**. ✓

A chave é:

- **Traces (Tempo)**: teu app no IntelliJ (host) envia OTLP → `localhost:4317` (porta publicada do container do Tempo).
- **Logs (Loki)**: teu app no IntelliJ escreve logs **em arquivo** num diretório do projeto (host) e o **Promtail em Docker** lê esse diretório via **bind mount** e empurra pro Loki.
- **Métricas (Prometheus)**: opção mais “sem dor de rede” é rodar o Prometheus em **host network** (só no Linux), e ele raspa `localhost:809x` direto.

Aí você consegue operar tudo pelo Grafana **enquanto desenvolve no IntelliJ**, sem containerizar os micros ainda.

## 1) TRACES do IntelliJ (host) → Tempo (Docker)

Você já tem o Tempo expondo `4317` . Então, em cada micro no IntelliJ:

VM options

```
-javaagent:/caminho/opentelemetry-javaagent.jar
```

## Env vars (Run configuration)

```
OTEL_SERVICE_NAME=ms-checkout-orchestrator # muda por micro
OTEL_EXPORTER_OTLP_ENDPOINT=http://localhost:4317 OTEL_EXPORTER_OTLP_PROTOCOL=grpc
OTEL_TRACES_EXPORTER=otlp OTEL_METRICS_EXPORTER=none OTEL_LOGS_EXPORTER=none
OTEL_RESOURCE_ATTRIBUTES=deployment.environment=local
```

✓ Resultado: traces aparecem no **Grafana** → **Tempo** mesmo com app rodando no host.

## 2) LOGS do IntelliJ (host) → Loki (Docker) sem instalar nada

Aqui é o “pulo do gato”: Loki não coleta stdout do IntelliJ. Então você faz:

1. **Logback** grava JSON em arquivo (no host)
2. **Promtail no Docker** faz tail desse arquivo via volume montado

### 2.1 Crie um diretório único de logs no seu repo

Na raiz do projeto (mesmo nível de `ms-*`), crie:

```
mkdir -p logs
```

Você vai ter:

- `logs/ms-order.json`
- `logs/ms-payment.json`
- etc.



## 2.2 Ajuste `logback-spring.xml` de cada micro (console + arquivo)

O Codex já criou JSON no console. Agora você adiciona um **File appender** apontando pra `../logs/...` (ou um path configurável).

Padrão que recomendo (fácil e consistente):

- Cada micro escreve em: `../logs/${spring.application.name}.json`

No `logback`:

- pegar `spring.application.name`
- pegar um `retail.logs.dir` (default `../logs`)

Se você quiser, eu te passo um `logback-spring.xml` completo “padrão ouro” com RollingFile + JSON + MDC. Mas o conceito é esse: **console + file**.

## 2.3 Monte esse diretório no Promtail (Docker)

No seu `containers/docker-compose.observability.yaml`, no serviço `promtail`, adicione:

```
promtail: volumes: - ../logs:/var/log/retail-store:ro -  
promtail_positions:/tmp/positions
```

(O `../logs` é relativo à pasta `containers/`, então ele aponta pro `logs/` da raiz do repo.)

E defina `promtail_positions` como named volume:

```
volumes: promtail_positions:
```

## 2.4 Promtail config: ler esses arquivos e labelar por serviço

No `containers/observability/promtail/promtail.yaml`, adicione um `scrape_config` assim:

```
scrape_configs: - job_name: retail-store-host-logs static_configs: - targets:
[localhost] labels: job: retail-store __path__: /var/log/retail-store/*.json
pipeline_stages: - json: expressions: level: level service: service trace_id: trace_id
span_id: span_id correlation_id: correlation_id order_id: order_id msg: message # se no
JSON não tiver "service", extraímos pelo nome do arquivo: ms-order.json -> service=ms-
order - regex: source: filename expression: ".*/(?P<svc>ms-[^/]+)\.json" - labels:
svc:
```

E o “truque” pra ter `filename` como label: o promtail costuma incluir `filename` quando tailing arquivo. Se não vier, a gente resolve via `relabel_config` usando `__path__` (eu te ajudo se precisar).

✅ Resultado: no Loki você busca por `{svc="ms-order"}` ou `{job="retail-store"}` e acha logs dos apps rodando no IntelliJ.

### 3) MÉTRICAS do IntelliJ (host) → Prometheus (Docker) sem treta de firewall

Você está no Ubuntu e já sofreu com `host.docker.internal` + UFW + subnets.

A forma mais “zero dor” no Linux é:

**Opção recomendada: Prometheus com `network_mode: host`**

No `containers/docker-compose.observability.yaml`, no `prometheus`:

```
prometheus: network_mode: host # remova "ports" e "networks" daqui
```

E no `prometheus.yml`, configure targets como:

```
- job_name: ms-order metrics_path: /actuator/prometheus static_configs: - targets:
['localhost:8091'] - job_name: ms-payment metrics_path: /actuator/prometheus
static_configs: - targets: ['localhost:8094'] ...
```

E pros exporters (você já publicou portas no host):

- kafka\_exporter → localhost:9308
- postgres\_exporter → localhost:9187

✓ Isso elimina totalmente o problema de container → host, porque o Prometheus passa a “ser o host” na rede.

Observação: com `network_mode: host`, o Prometheus **não usa DNS do Docker**. Então no `prometheus.yml` você sempre aponta para `localhost:PORT` (usando as portas publicadas).

## 4) O que você ganha com isso (o que você quer de verdade)

Com esse setup:

- ✓ Você roda `ms-order`, `ms-payment`, `ms-checkout-orchestrator`, `ms-inventory` no IntelliJ
- ✓ Vê **métricas** no Grafana (via Prometheus)
- ✓ Vê **logs** no Grafana (via Loki + Promtail lendo arquivos do host)
- ✓ Vê **traces ponta a ponta** no Grafana (via Tempo recebendo OTLP do host)
- ✓ Clica no `TraceID` do log e abre o trace

## 5) Alternativa (se você odiar log em arquivo): mandar log direto pro Loki

Existe um caminho “sem promtail”: usar um appender do logback que empurra pro Loki (tipo Loki4j).

Prós: zero arquivo, zero tail

Contras: mais dependência/config dentro de cada micro

Eu só recomendo depois, quando você quiser “ficar elegante”. Para agora, **arquivo + promtail** é simples e muito robusto.

### Se você quiser, eu já te deixo uma lista de tarefas pro Codex (bem objetiva)

Me diga só:

1. Você quer adotar **Prometheus network\_mode: host** (eu recomendo)?
2. Você quer logs em `repo-root/logs/*.json` (recomendo)?

Se você disser “sim/sim”, eu te devolvo um checklist **cirúrgico** com:

- patch exato no `docker-compose observability.yaml`
- patch exato no `promtail.yaml`
- patch padrão no `logback-spring.xml`
- e as env vars do IntelliJ por micro (nome certinho).

sim, eu quero

Fechou. Aqui está o plano **cirúrgico** (com patches claros) pra você ter **logs + traces + métricas** dos micros rodando no **IntelliJ (host)**, com **toda a infra de observabilidade em Docker**, sem instalar nada no Ubuntu.

## 0) Estrutura esperada

Na raiz do repo (mesmo nível de `ms-*` e `containers/`), crie:

```
mkdir -p logs
```

Você terá arquivos tipo:

- `logs/ms-order.json`
- `logs/ms-payment.json`
- `logs/ms-checkout-orchestrator.json`
- `logs/ms-inventory.json`
- etc.

## 1) Prometheus “sem dor”: `network_mode: host`

Como você está no Linux, esse é o jeito mais robusto de raspar `localhost:809x` dos apps rodando no IntelliJ.

### 1.1 Patch no `containers/docker-compose.observability.yaml`

No serviço `prometheus`, faça:

- adicionar:

`network_mode: host`

- remover do prometheus:
  - ports: (não precisa, host network já expõe)
  - networks: (host network não participa de bridges)
  - qualquer referência a DNS interno (tipo `loki:3100` , etc. não afeta prometheus)

✓ Depois disso, você acessa Prometheus em:

- `http://localhost:9090` (igual)

## 1.2 Patch no `containers/observability/prometheus/prometheus.yml`

Alvos devem ser `localhost:PORT` (no host):

Exemplo:

```
scrape_configs: - job_name: prometheus static_configs: - targets: ['localhost:9090'] -
job_name: kafka_exporter static_configs: - targets: ['localhost:9308'] - job_name:
postgres_exporter static_configs: - targets: ['localhost:9187'] - job_name: ms-order
metrics_path: /actuator/prometheus static_configs: - targets: ['localhost:8091'] -
job_name: ms-invoice metrics_path: /actuator/prometheus static_configs: - targets:
['localhost:8092'] - job_name: ms-notification metrics_path: /actuator/prometheus
static_configs: - targets: ['localhost:8093'] - job_name: ms-payment metrics_path:
/actuator/prometheus static_configs: - targets: ['localhost:8094'] - job_name: ms-
shipping metrics_path: /actuator/prometheus static_configs: - targets:
['localhost:8095'] - job_name: ms-customer metrics_path: /actuator/prometheus
static_configs: - targets: ['localhost:8096'] - job_name: ms-checkout-orchestrator
metrics_path: /actuator/prometheus static_configs: - targets: ['localhost:8097'] -
job_name: ms-inventory metrics_path: /actuator/prometheus static_configs: - targets:
['localhost:8098']
```

✓ Resultado: acabou a treta de UFW/subnet/host.docker.internal.

## 2) Logs do IntelliJ → Loki via Promtail em Docker (com bind mount)

### 2.1 Patch no containers/docker-compose.observability.yaml (promtail)

No serviço `promtail`, adicione o bind mount do diretório `logs/`:

Você roda `compose` dentro de `containers/`, então `../logs` aponta para `repo-root/logs`.

```
promtail: volumes: - ../observability/promtail/promtail.yaml:/etc/promtail/config.yaml:ro
- ../logs:/var/log/retail-store:ro - promtail_positions:/tmp/positions
```

E declare o volume no fim do arquivo:

```
volumes: promtail_positions:
```

✓ Resultado: promtail vê os logs do host sem instalar nada.

### 2.2 Patch no containers/observability/promtail/promtail.yaml

Adicione um `scrape_config` para esses arquivos:

```
scrape_configs: - job_name: retail-store-host-logs static_configs: - targets:
[localhost] labels: job: retail-store __path__: /var/log/retail-store/*.json
pipeline_stages: - json: expressions: level: level service: service trace_id: trace_id
span_id: span_id correlation_id: correlation_id order_id: order_id message: message #
se o JSON não tiver "service", tenta derivar pelo nome do arquivo - regex: source:
filename expression: ".*/(?P<svc>ms-[^/]+)\\.json" - labels: svc:
```

Se o `filename` não vier automaticamente, eu te dou o ajuste com `relabel_configs` (é rápido). Mas na maioria dos casos o `promtail` fornece.

### 3) Logback: console + arquivo (JSON) em TODOS os micros

Você quer acompanhar no IntelliJ e também mandar pro Loki.

#### 3.1 Em cada micro: `src/main/resources/logback-spring.xml`

Garanta 2 appenders:

- Console JSON
- File JSON para `../logs/<service>.json`

O caminho mais limpo é via property:

- `RETAIL_LOGS_DIR` default: `../logs`
- nome do arquivo: `${spring.application.name}.json`

Exemplo (essência do que precisa existir):

- File appender para `${LOG_DIR}/${spring.application.name}.json`

✅ Resultado: rodando no IntelliJ, cada micro escreve no seu arquivo.

### 4) Traces do IntelliJ → Tempo (Docker) com OTEL Java Agent

Você já tem o Tempo publicando 4317 .

#### 4.1 IntelliJ (por micro)



## VM options

```
-javaagent:/caminho/opentelemetry-javaagent.jar
```

## Env vars

```
OTEL_SERVICE_NAME=ms-order # muda por micro
OTEL_EXPORTER_OTLP_ENDPOINT=http://localhost:4317 OTEL_EXPORTER_OTLP_PROTOCOL=grpc
OTEL_TRACES_EXPORTER=otlp OTEL_METRICS_EXPORTER=none OTEL_LOGS_EXPORTER=none
OTEL_RESOURCE_ATTRIBUTES=deployment.environment=local
```

✓ Resultado: traces aparecem no Tempo.

## 5) Subida e validação

### 5.1 Suba a infra

```
cd containers docker compose -f docker-compose.yaml -f docker-
compose/observability.yaml down --remove-orphans docker compose -f docker-compose.yaml
-f docker-compose/observability.yaml up -d
```

### 5.2 Rode 1 micro no IntelliJ e valide

#### Métricas

- `http://localhost:9090/targets` → job do micro deve ficar **UP**

#### Logs

- Grafana → Explore → Loki

- Query:

```
{job="retail-store"}
```

ou, se o label `svc` estiver funcionando:

```
{svc="ms-order"}
```

## Traces

- Grafana → Explore → Tempo
- e/ou clique no `TraceID` no log (derived field) para abrir o trace.

## 6) Mensagem pronta pro Codex (pra implementar certinho)

Copia e cola:

```
Implementar observabilidade para micros rodando no IntelliJ (host), mantendo infra em Docker e sem instalar nada no Ubuntu. 1) Prometheus: - Alterar containers/docker-compose/observability.yaml: setar prometheus com network_mode: host. - Remover ports/networks do prometheus. - Atualizar containers/observability/prometheus/prometheus.yml: targets devem ser localhost:8091..8098 (actuator/prometheus), e exporters em localhost:9187/9308. 2) Logs host -> Loki: - Criar diretório repo-root/logs (se não existir). - Alterar promtail no docker-compose/observability.yaml para montar ../logs em /var/log/retail-store:ro e usar um volume named promtail_positions. - Atualizar containers/observability/promtail/promtail.yaml: adicionar scrape_config que lê /var/log/retail-store/*.json e parseia JSON (trace_id/span_id/order_id/correlation_id) e cria labels job=retail-store e, se possível, svc derivado do filename ms-*.json. 3) Logback em todos os micros: - Em ms-*/src/main/resources/logback-spring.xml: manter
```

console JSON e adicionar file JSON appender para  
../logs/\${spring.application.name}.json (dir configurável). - Garantir que o JSON  
inclua trace\_id/span\_id e MDC fields (order\_id/correlation\_id/saga\_id etc.) 4) Traces:  
- Documentar env vars do IntelliJ para OTel Java Agent exportar para  
http://localhost:4317 (Tempo). 5) Validar: - Subir compose (base + observability). -  
Rodar 1 micro no IntelliJ e confirmar: Prometheus targets UP, logs aparecendo em Loki,  
click TraceID abrindo trace no Tempo.

Se você quiser, eu também te passo:

- um logback-spring.xml pronto “padrão ouro” (com rolling + JSON + MDC completo),
- e uma query Loki “filme do pedido” usando order\_id / correlation\_id .

sim

Fechado — aqui vai um logback-spring.xml **padrão-ouro** (console + arquivo JSON, com MDC completo,  
pronto pra Promtail/Loki e com trace\_id/span\_id ), e as **queries Loki “filme do pedido”**.

**Objetivo:** você roda no IntelliJ (host), vê logs no console, e ao mesmo tempo grava  
em ../logs/<service>.json para o Promtail (em Docker) coletar e mandar pro Loki.

## 1) logback-spring.xml padrão-ouro (Spring Boot 3.x)

Coloque este arquivo em **todos** os micros:

ms-\*/src/main/resources/logback-spring.xml

```
<?xml version="1.0" encoding="UTF-8"?> <configuration scan="false"> <!-- ===== LOG
DIR ===== Para apps rodando no IntelliJ (host), isso cria arquivos em ../logs/ na
raiz do repo. Você pode sobrescrever com env var: RETAIL_LOGS_DIR=/algum/path -->
<springProperty scope="context" name="APP_NAME" source="spring.application.name"
defaultValue="unknown-service"/> <property name="LOG_DIR"
value="${RETAIL_LOGS_DIR:-../logs}"/> <property name="LOG_FILE"
value="${LOG_DIR}/${APP_NAME}.json"/> <!-- ===== ENCODERS ===== --> <appender
name="CONSOLE_JSON" class="ch.qos.logback.core.ConsoleAppender"> <encoder
class="net.logstash.logback.encoder.LoggingEventCompositeJsonEncoder"> <providers> <!--
timestamp --> <timestamp> <fieldName>@timestamp</fieldName> </timestamp> <!-- log level
/ logger / thread --> <logLevel> <fieldName>level</fieldName> </logLevel> <loggerName>
<fieldName>logger</fieldName> </loggerName> <threadName> <fieldName>thread</fieldName>
</threadName> <!-- message --> <message> <fieldName>message</fieldName> </message> <!--
throwable stacktrace --> <stackTrace> <fieldName>exception</fieldName> </stackTrace>
<!-- service name --> <globalCustomFields>{"service":"${APP_NAME}","env":"local"}
</globalCustomFields> <!-- MDC: OpenTelemetry + negócio --> <mdc> <!-- inclui tudo do
MDC; se quiser restringir, troque por includeMdcKeyName --> </mdc> </providers>
</encoder> </appender> <!-- Arquivo JSON (para Promtail ler e mandar pro Loki) -->
<appender name="FILE_JSON" class="ch.qos.logback.core.rolling.RollingFileAppender">
<file>${LOG_FILE}</file> <!-- Rolling por tamanho + histórico (bom para dev) -->
<rollingPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedRollingPolicy">
<fileNamePattern>${LOG_DIR}/${APP_NAME}_%d{yyyy-MM-dd}_%i.json</fileNamePattern>
<maxFileSize>20MB</maxFileSize> <maxHistory>7</maxHistory>
<totalSizeCap>1GB</totalSizeCap> </rollingPolicy> <encoder
class="net.logstash.logback.encoder.LoggingEventCompositeJsonEncoder"> <providers>
<timestamp> <fieldName>@timestamp</fieldName> </timestamp> <logLevel>
<fieldName>level</fieldName> </logLevel> <loggerName> <fieldName>logger</fieldName>
</loggerName> <threadName> <fieldName>thread</fieldName> </threadName> <message>
<fieldName>message</fieldName> </message> <stackTrace> <fieldName>exception</fieldName>
</stackTrace> <globalCustomFields>{"service":"${APP_NAME}","env":"local"}
```

```
</globalCustomFields> <!-- Puxa MDC (inclui trace_id/span_id do OTel agent e seus campos: correlation_id, order_id, saga_id...) --> <mdc/> </providers> </encoder>
</appender> <!-- ===== LEVELS ===== --> <root level="INFO"> <appender-ref
ref="CONSOLE_JSON"/> <appender-ref ref="FILE_JSON"/> </root> <!-- Ajustes úteis -->
<logger name="org.springframework" level="INFO"/> <logger name="org.apache.kafka"
level="INFO"/> <logger name="org.hibernate" level="WARN"/> </configuration>
```

## O que esse arquivo te dá

- Console JSON (bom no IntelliJ)
- Arquivo JSON por serviço (bom pro Promtail)
- Inclui automaticamente **MDC todo**:
  - trace\_id , span\_id , trace\_flags (OTel agent)
  - correlation\_id , parent\_correlation\_id , order\_id , saga\_id , saga\_step etc (se você já colocou no MDC)

✅ Isso é o mínimo “profissional” pra Loki + Tempo se encaixarem bem.

## 2) Onde garantir que order\_id / correlation\_id entram no MDC

Como você já tem interceptors/filtros, só confirme que o MDC está usando chaves consistentes, por exemplo:

- order\_id
- correlation\_id
- parent\_correlation\_id
- saga\_id
- saga\_step

(Se hoje você usa outro nome, a gente padroniza depois.)

### 3) Queries Loki “filme do pedido” (copiar e colar)

#### 3.1 Tudo de um pedido (OrderId)

```
{job="retail-store"} | json | order_id="SEU_ORDER_ID"
```

#### 3.2 Só erros do pedido

```
{job="retail-store"} | json | order_id="SEU_ORDER_ID" | level="ERROR"
```

#### 3.3 Tudo da saga

```
{job="retail-store"} | json | saga_id="SEU_SAGA_ID"
```

#### 3.4 Só o checkout “saga step”

```
{job="retail-store"} | json | service="ms-checkout-orchestrator" |  
saga_step="WAIT_PAYMENT_CAPTURE"
```

#### 3.5 Encontrar pelo correlation id (quando ainda não tem order id)

```
{job="retail-store"} | json | correlation_id="SEU_CORRELATION_ID"
```

Quando você abrir uma linha, deve aparecer `trace_id`. Aí você clica no **TraceID** (derived field) e abre o trace no Tempo.

## 4) Dica rápida pra evitar dor de cabeça

Se por acaso o Promtail não conseguir “parsear” o JSON por causa de campos grandes ou stacktrace, tudo bem: ele ainda manda como texto. Mas pra ficar redondo, mantenha logs como JSON “flat” (o encoder já faz isso).

## 5) Quer que eu te mande também o `promtail.yaml` completo já com labels `svc` por filename?

Se você me disser como você quer nomear os arquivos (ex.: `ms-order.json` mesmo), eu te passo o `promtail.yaml` final com:

- `labels svc=ms-order`
- `parse de trace_id e order_id`
- `e query pronta {svc="ms-order"}`

Só me diga: **o nome do arquivo vai ser exatamente** `${spring.application.name}.json` (ex.: `ms-order.json`)?

sei nao , sugira voce.... deixa uma orientação para o codex

Beleza — vou sugerir um padrão **simples, estável e “à prova de Codex”**, e já deixo uma orientação completa pra ele implementar sem quebrar nada.

## Padrão recomendado (minha sugestão)

### Nome do arquivo de log por micro

✓ Use o nome do serviço (spring.application.name) como base:

- logs/ms-order.json
- logs/ms-payment.json
- logs/ms-checkout-orchestrator.json
- logs/ms-inventory.json
- etc.

### Por quê?

- não depende de PID nem de porta
- não muda entre máquina, IDE, ou ambiente
- fica fácil de filtrar no Loki por “service”

### Diretório único no repo

✓ Sempre gravar em repo-root/logs/ (um diretório só), para o Promtail coletar tudo com um único mount.



## Orientação “perfeita” para o Codex (copiar/colar)

Objetivo: coletar logs e traces de microserviços rodando NO HOST (IntelliJ) usando a stack de observabilidade em Docker (Grafana/Loki/Tempo/Prometheus), sem instalar nada no Ubuntu. PADRÃO DE LOGS (host -> Loki): 1) Criar diretório na raiz do repo: ./logs 2) Em TODOS os microserviços, ajustar src/main/resources/logback-spring.xml para: - continuar logando JSON no console (para o IntelliJ) - também escrever JSON em arquivo em ../logs/\${spring.application.name}.json - permitir override do diretório via env var RETAIL\_LOGS\_DIR (default ../logs) - incluir MDC completo (trace\_id/span\_id + correlation\_id/order\_id/saga\_id/saga\_step etc) 3) No containers/docker-compose.observability.yaml: - no serviço promtail, montar o diretório do host: - ../logs:/var/log/retail-store:ro - adicionar volume named para positions: - promtail\_positions:/tmp/positions - declarar volumes: promtail\_positions 4) Em containers/observability/promtail/promtail.yaml: - adicionar um scrape\_config para /var/log/retail-store/\*.json - parsear JSON com pipeline\_stages (json -> expressions) - garantir label estável para o serviço: - preferencialmente usar o campo "service" do próprio JSON (que vem do logback globalCustomFields) - adicionalmente, derivar label svc pelo nome do arquivo: ms-\*.json (fallback) - resultado esperado: no Loki, queries como {job="retail-store", service="ms-order"} funcionam TRACES (host -> Tempo): 5) Documentar Run Configuration do IntelliJ (por micro): - VM Options: - javaagent:/path/opentelemetry-javaagent.jar - Env: OTEL\_SERVICE\_NAME=<spring.application.name> OTEL\_EXPORTER\_OTLP\_ENDPOINT=http://localhost:4317 OTEL\_EXPORTER\_OTLP\_PROTOCOL=grpc OTEL\_TRACES\_EXPORTER=otlp OTEL\_METRICS\_EXPORTER=none OTEL\_LOGS\_EXPORTER=none OTEL\_RESOURCE\_ATTRIBUTES=deployment.environment=local MÉTRICAS (host -> Prometheus): 6) Para evitar problemas de rede Docker->host no Linux: - configurar o prometheus no compose com network\_mode: host - atualizar prometheus.yml para raspar localhost:8091..8098 (actuator/prometheus) - exporters devem ser localhost:9187 e localhost:9308 - remover ports/networks do prometheus ao usar host network VALIDAÇÃO: 7) Subir infra: docker compose -f docker-compose.yaml -f docker-compose.observability.yaml up -d 8) Rodar um micro no IntelliJ e validar: - /actuator/prometheus responde no host - Prometheus targets UP - Loki recebe logs do

arquivo (query {job="retail-store"} | json) - clicar no TraceID do log abre o trace no Tempo

## Dicas (pra evitar cagada do Codex)

- **Não** colocar logs em /tmp (apaga, muda, dá dor de cabeça).
- **Não** usar nome com PID no arquivo (dificulta query e histórico).
- **Não** depender de host.docker.internal para Prometheus no Linux (use network\_mode: host ).
- **Sempre** manter service=<spring.application.name> no JSON (globalCustomFields), porque isso vira label/consulta no Loki.

Se você quiser, eu também posso te passar um promtail.yaml completo já no formato ideal (com json pipeline + labels bem certinhas), mas com essa orientação o Codex já consegue implementar de forma bem fluida.

sim

Beleza — aqui vai um promtail.yaml **completo, padrão-ouro**, pronto pra:

- ler repo-root/logs/\*.json (logs gerados pelos micros rodando no IntelliJ)
- parsear JSON (pra você filtrar por service , level , order\_id , etc.)
- criar labels úteis ( job , service , env , level )

- manter `trace_id` no conteúdo do log (pra **derived field** no Grafana abrir o Tempo)

**Importante:** esse arquivo assume que o seu `logback-spring.xml` inclui:

- `{"service":"${APP_NAME}", "env":"local"}` em `globalCustomFields`
- e que o log é JSON (logstash encoder)

Se ainda não estiver assim, peça pro Codex alinhar (ele já estava no caminho).

## 1) containers/observability/promtail/promtail.yaml (completo)

```
server: http_listen_port: 9080 grpc_listen_port: 0 positions: filename:
/tmp/positions/positions.yaml clients: - url: http://loki:3100/loki/api/v1/push
scrape_configs: # ===== # Logs dos
microserviços rodando NO HOST (IntelliJ) # Arquivos em repo-root/logs/*.json montados
em: # /var/log/retail-store/*.json #
===== - job_name: retail-store-
host-logs static_configs: - targets: [localhost] labels: job: retail-store __path__:
/var/log/retail-store/*.json pipeline_stages: # 1) Parse JSON do logback - json:
expressions: ts: "@timestamp" level: "level" logger: "logger" thread: "thread" message:
"message" service: "service" env: "env" # OpenTelemetry MDC (quando existir) trace_id:
"trace_id" span_id: "span_id" trace_flags: "trace_flags" # Negócio / correlação (quando
existir) order_id: "order_id" correlation_id: "correlation_id" parent_correlation_id:
"parent_correlation_id" saga_id: "saga_id" saga_step: "saga_step" aggregate_id:
"aggregate_id" aggregate_type: "aggregate_type" event_type: "event_type" command_type:
"command_type" # 2) Normalize timestamp (se existir @timestamp) - timestamp: source: ts
format: RFC3339Nano # 3) Promove alguns campos para labels (cuidado para não explodir
cardinalidade) # Labels bons: service/env/level (baixa cardinalidade) - labels:
service: env: level: # 4) Ajusta o output do log para ficar legível e ainda manter
campos no JSON # Você pode mudar para output: source: message se quiser só a mensagem.
- output: source: message
```

## Por que eu NÃO coloquei `order_id` como label?

Porque isso explode cardinalidade no Loki (cada pedido vira uma label distinta).

O melhor é manter `order_id` como **campo** e filtrar com `| json | order_id="..."`.

## 2) Ajustes no `docker-compose observability.yaml` (promtail)

O promtail precisa:

- montar a config
- montar o diretório `../logs` do host
- persistir positions num named volume

Exemplo do bloco (o Codex deve adaptar ao seu arquivo):

```
promtail: image: grafana/promtail:latest command: -config.file=/etc/promtail/config.yml
volumes: - ../observability/promtail/promtail.yaml:/etc/promtail/config.yml:ro -
  ../logs:/var/log/retail-store:ro - promtail_positions:/tmp/positions depends_on: - loki
networks: - retail-store
```

E no final do compose:

```
volumes: promtail_positions:
```

## 3) Como consultar no Loki (queries prontas)

Tudo de um serviço:

```
{job="retail-store", service="ms-order"}
```

### Tudo de um pedido (campo JSON):

```
{job="retail-store"} | json | order_id="SEU_ORDER_ID"
```

### Erros do pedido:

```
{job="retail-store"} | json | order_id="SEU_ORDER_ID" | level="ERROR"
```

### Só saga do checkout:

```
{job="retail-store", service="ms-checkout-orchestrator"} | json | saga_id="SEU_SAGA_ID"
```

## 4) Checklist rápido pra você validar em 2 minutos

1. Suba infra:

```
cd containers docker compose -f docker-compose.yaml -f docker-  
compose.observability.yaml up -d
```

2. Rode um micro no IntelliJ e confirme que ele escreve no arquivo:

```
ls -l logs/ tail -n 3 logs/ms-checkout-orchestrator.json
```

3. Veja se o promtail está enviando:

```
docker logs --tail 50 containers-promtail-1
```

4. Grafana → Explore → Loki:

```
{job="retail-store"}
```

Se aparecer, fechou.

## 5) Orientação final pro Codex (pra não errar)

- **Não** colocar `order_id` como label no Loki.
- Labels no Loki: `job`, `service`, `env`, `level` (só).
- `order_id/correlation_id/saga_id` ficam no JSON e filtram via `| json`.
- Garantir que `service` e `env` estejam no JSON via `globalCustomFields` no logback.

Se você quiser, eu também te passo um `docker-compose.observability.yaml` “diff” do prometheus com `network_mode: host` + ajustes de exporters no `prometheus.yml` (pra ficar 100% alinhado ao seu setup dev com IntelliJ).

sim

Pensou por ...

Fechado — vou te passar o **diff “padrão ouro”** (bem direto) para:

1. **Prometheus em** `network_mode: host` (no Linux) pra raspar `localhost:809x` dos micros rodando no IntelliJ
2. Ajustar o `prometheus.yml` pra usar `localhost`
3. Ajustar o **Grafana datasource do Prometheus**, porque quando o Prometheus sai da rede do Compose, o Grafana **não consegue mais** falar com ele via `http://prometheus:9090` (DNS do compose).

**Regra respeitada:** não mexe no `containers/docker-compose.yaml` (o que já funciona). Só no `docker-compose.observability.yaml` e configs da pasta `observability/`.

## 1) Patch: `containers/docker-compose.observability.yaml` (Prometheus em host)

### ✓ Objetivo

- Prometheus no host network
- sem `ports`: (não precisa)
- sem `networks`: (host network não participa)
- (opcional) aumentar um pouco timeouts do scrape via config

### Diff sugerido (conceitual)

Procura o bloco do `prometheus`: e muda para algo assim:

```
services: prometheus: image: prom/prometheus:latest command: - "--
config.file=/etc/prometheus/prometheus.yml" volumes: -
./observability/prometheus/prometheus.yml:/etc/prometheus/prometheus.yml:ro -
prometheus_data:/prometheus # >>> ADD (Linux): Prometheus no host network pra raspar
```

```
localhost:809x network_mode: host # >>> REMOVE: ports (host network já expõe) # ports:
# - "9090:9090" # >>> REMOVE: networks (host network não usa) # networks: # - retail-
store
```

⚠ **Detalhe importante:** com `network_mode: host`, o Prometheus deixa de existir como “service” dentro do DNS do Compose. Ou seja: `http://prometheus:9090` **quebra** para outros containers (Grafana). Por isso vem o patch do Grafana logo abaixo.

2)

## Patch: containers/observability/prometheus/prometheus.yml (tar gets localhost)

### ✓ Objetivo

- Scrape dos micros no host: `localhost:8091..8098`
- Exporters via portas publicadas no host:
  - `localhost:9187` (postgres\_exporter)
  - `localhost:9308` (kafka\_exporter)

Exemplo completo do trecho (mantendo seu estilo):

```
global: scrape_interval: 10s scrape_timeout: 10s evaluation_interval: 10s
scrape_configs: - job_name: prometheus static_configs: - targets: ["localhost:9090"] -
job_name: kafka_exporter static_configs: - targets: ["localhost:9308"] - job_name:
postgres_exporter static_configs: - targets: ["localhost:9187"] - job_name: ms-order
metrics_path: /actuator/prometheus static_configs: - targets: ["localhost:8091"] -
job_name: ms-invoice metrics_path: /actuator/prometheus static_configs: - targets:
["localhost:8092"] - job_name: ms-notification metrics_path: /actuator/prometheus
static_configs: - targets: ["localhost:8093"] - job_name: ms-payment metrics_path:
```



```
/actuator/prometheus static_configs: - targets: ["localhost:8094"] - job_name: ms-  
shipping metrics_path: /actuator/prometheus static_configs: - targets:  
["localhost:8095"] - job_name: ms-customer metrics_path: /actuator/prometheus  
static_configs: - targets: ["localhost:8096"] - job_name: ms-checkout-orchestrator  
metrics_path: /actuator/prometheus static_configs: - targets: ["localhost:8097"] -  
job_name: ms-inventory metrics_path: /actuator/prometheus static_configs: - targets:  
["localhost:8098"]
```

✓ Resultado: Prometheus enxerga os apps do IntelliJ sem `host.docker.internal`, sem UFW/subnet gambiarra.

### 3) Patch: Grafana datasource do Prometheus (porque prometheus:9090 não existe mais)

Você deve ter um arquivo tipo:

```
containers/observability/grafana/provisioning/datasources/datasources.yaml
```

Normalmente ele está com algo assim:

```
- name: Prometheus type: prometheus url: http://prometheus:9090
```

Isso vai falhar agora.

✓ **Solução recomendada: usar `host.docker.internal:9090`**

No Linux, você precisa garantir que o container do Grafana resolve `host.docker.internal` → host gateway.

#### 3.1 Patch no `docker-compose.observability.yaml` (Grafana)

No serviço grafana , adicione:

```
grafana: extra_hosts: - "host.docker.internal:host-gateway"
```

### 3.2 Patch no datasource do Grafana

Troque a URL do Prometheus para:

```
url: http://host.docker.internal:9090
```

Exemplo:

```
apiVersion: 1 datasources: - name: Prometheus type: prometheus access: proxy url:
http://host.docker.internal:9090 isDefault: true
```

✅ Resultado: Grafana (container) fala com Prometheus (no host network) via host gateway.

## 4) Subir e validar (checklist rápido)

1. Reiniciar stack:

```
cd containers docker compose -f docker-compose.yaml -f docker-
compose.observability.yaml down --remove-orphans docker compose -f docker-compose.yaml
-f docker-compose.observability.yaml up -d
```

2. Validar Prometheus:

- `http://localhost:9090/targets`

Deve ficar **UP** para os micros que estiverem rodando no IntelliJ.

### 3. Validar Grafana datasource:

- Grafana → Connections → Data sources → Prometheus → "Save & test"
- deve dar 

## 5) Se ainda der erro de conexão do Grafana → Prometheus

Como você tem UFW ativo, pode bloquear a conexão vinda da bridge do Compose para porta 9090 no host. Se acontecer, libere a subnet do Compose (a mesma lógica de antes).

Descobre o gateway/subnet (mais direto):

```
docker network ls | docker network inspect <network_do_compose> | grep -A2 Subnet
```

Aí libera, por exemplo:

```
sudo ufw allow from 172.18.0.0/16 to any port 9090 proto tcp
```

## Orientação final pro Codex (bem explícita)

- Prometheus em host network = ótimo para raspar apps do IntelliJ
- Mas precisa ajustar o datasource do Grafana para `host.docker.internal:9090` e adicionar `extra_hosts` no Grafana
- Não usar mais `host.docker.internal:809x` no Prometheus (vai ser `localhost:809x`)
- Exporters continuam via `localhost:9187/9308` (porque você já publica portas no host)

Se você quiser, eu também te escrevo um “script de validação” (3 curls + 1 query Loki) pra você rodar e ter certeza que métricas/logs/traces estão 100% integrados.

sim

Beleza — aqui vai um **script de validação “tanque de guerra”** pra você rodar e confirmar **MÉTRICAS + LOGS + TRACES** (end-to-end) no seu setup **IntelliJ no host + observability em Docker**.

Ele não instala nada. Só usa `bash`, `docker`, `curl` (que você já tem) e checa tudo em ordem.

## Script: `scripts/validate-observability.sh`

Crie `scripts/validate-observability.sh` na raiz do repo:

```
#!/usr/bin/env bash set -euo pipefail echo "== Retail Store Observability Validation
==" # ----- # Config # ----- SERVICES=( "ms-order:8091" "ms-invoice:8092" "ms-
notification:8093" "ms-payment:8094" "ms-shipping:8095" "ms-customer:8096" "ms-
checkout-orchestrator:8097" "ms-inventory:8098" ) echo echo "1) Checking Docker
containers (Grafana/Loki/Tempo/Promtail/Prometheus)..." docker ps --format 'table
{{.Names}}\t{{.Status}}\t{{.Ports}}' | egrep 'grafana|loki|tempo|promtail|prometheus'
|| { echo "ERROR: Observability containers not found/running." exit 1 } echo echo "2)
Checking Grafana/Loki/Tempo/Prometheus health endpoints..." echo -n "Grafana : "; curl
-s -o /dev/null -w "%{http_code}\n" http://localhost:3000/login echo -n "Loki : "; curl
-s http://localhost:3100/ready || true; echo echo -n "Tempo : "; curl -s
http://localhost:3200/ready || true; echo echo -n "Prometheus: "; curl -s -o /dev/null
-w "%{http_code}\n" http://localhost:9090/-/ready echo echo "3) Checking Spring
```

```

Actuator on HOST (only for services that are currently running in IntelliJ)..." for s
in "${SERVICES[@]}"; do name="${s%:*}" port="${s##*:}" code="$(curl -s -o /dev/null -w
"%{http_code}" "http://localhost:${port}/actuator/health" || true)" if [[ "$code" ==
"200" ]]; then echo "OK ${name} health on :${port}" else echo "SKIP ${name} health on
:${port} (HTTP $code) - probably not running" fi done echo echo "4) Checking
/actuator/prometheus on HOST (must be 200 when service is running)..." for s in
"${SERVICES[@]}"; do name="${s%:*}" port="${s##*:}" code="$(curl -s -o /dev/null -w "%
{http_code}" "http://localhost:${port}/actuator/prometheus" || true)" if [[ "$code" ==
"200" ]]; then echo "OK ${name} prometheus endpoint on :${port}" else echo "SKIP
${name} prometheus endpoint on :${port} (HTTP $code)" fi done echo echo "5) Checking
Prometheus targets page is reachable (manual check recommended too)..." curl -s -o
/dev/null -w "Prometheus targets HTTP %{http_code}\n" http://localhost:9090/targets
echo echo "6) Checking that log files exist on HOST (repo-root/logs/*.json)..." if [[ -
d "./logs" ]]; then ls -lah ./logs | sed -n '1,40p' count="$(ls -1 ./logs/*.json
2>/dev/null | wc -l | tr -d ' ')" if [[ "$count" -ge 1 ]]; then echo "OK Found ${count}
JSON log file(s) in ./logs" else echo "WARN No *.json logs found in ./logs yet (start a
service + ensure logback writes to file)." fi else echo "ERROR ./logs directory not
found." exit 1 fi echo echo "7) Checking Promtail can see the mounted logs
directory..." # adjust promtail container name if different in your compose
PROMTAIL_CONTAINER="$(docker ps --format '{{.Names}}' | grep -E 'promtail' | head -n 1
|| true)" if [[ -z "${PROMTAIL_CONTAINER}" ]]; then echo "ERROR promtail container not
found." exit 1 fi echo "Promtail container: ${PROMTAIL_CONTAINER}" docker exec -it
"${PROMTAIL_CONTAINER}" sh -lc 'ls -lah /var/log/retail-store | sed -n "1,40p"' || {
echo "ERROR promtail cannot access /var/log/retail-store (mount missing?)" exit 1 }
echo echo "8) Basic Loki query check (via Loki HTTP API)..." # Note: Loki API uses
query_range; we do a minimal query that should succeed even if no logs yet. # We avoid
jq dependency; just check HTTP status.
LOKI_QUERY_URL="http://localhost:3100/loki/api/v1/query?query=%7Bjob%3D%22retail-
store%22%7D" code="$(curl -s -o /dev/null -w "%{http_code}" "${LOKI_QUERY_URL}" ||
true)" echo "Loki query HTTP $code (200 means Loki is responding)" if [[ "$code" !=
"200" ]]; then echo "WARN Loki query not OK. Check Loki container logs and promtail
config." fi echo echo "9) Trace pipeline sanity check:" echo "Start/trigger any request
in a running service (HTTP or Kafka), then:" echo "- open Grafana -> Explore -> Loki ->

```

```
query: {job=\"retail-store\"} | json" echo "- find a log line with trace_id and click  
TraceID to open Tempo trace." echo echo "DONE ✓"
```

Depois:

```
chmod +x scripts/validate-observability.sh ./scripts/validate-observability.sh
```

## O que esse script confirma

- ✓ Containers de observabilidade estão rodando
- ✓ Endpoints de health de Loki/Tempo/Prometheus respondem
- ✓ Apps no IntelliJ respondem /actuator/health e /actuator/prometheus
- ✓ ./logs/\*.json existe
- ✓ Promtail enxerga o diretório montado
- ✓ Loki responde ao endpoint de query

O único passo “manual” que sobra é o **E2E trace** (porque depende de você disparar uma requisição/evento), mas eu coloquei o workflow exato.

## Dica extra: 2 comandos “instantâneos” quando você disparar um pedido

### 1) Ver o “filme” pelo OrderId no Loki

```
{job="retail-store"} | json | order_id="SEU_ORDER_ID"
```

## 2) Ver o “filme” pelo correlation id

```
{job="retail-store"} | json | correlation_id="SEU_CORRELATION_ID"
```

Depois clica no **TracelD** e pronto: trace completo no Tempo.

Se você colar aqui a saída do `./scripts/validate-observability.sh`, eu te digo exatamente o que ainda falta (se faltar algo) e qual ajuste mínimo fazer.