

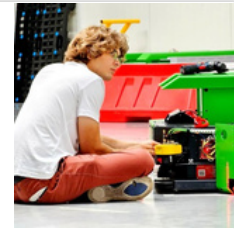
Kubernetes EKS: Containerize ANY, C#, Java, js framework, Python, Go, Rust, C++, PHP, Ruby based microservice app on Linux with OpenTelemetry Collector implementation

Check GitHub for helpful DevOps tools:

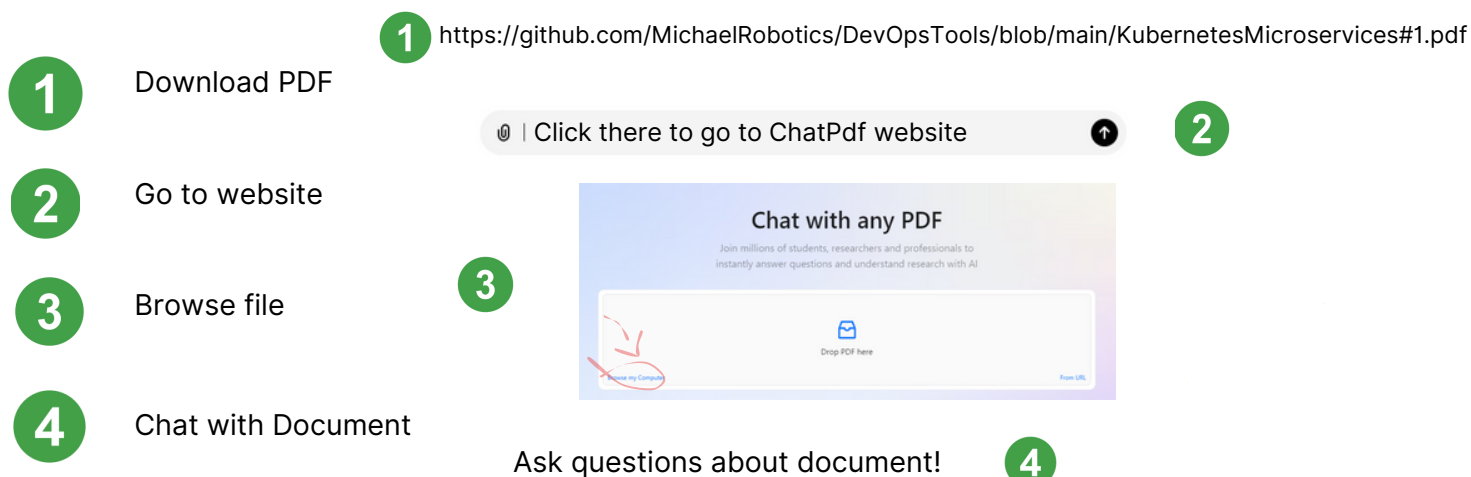
Michael Robotics

Hi, I'm Michal. I'm a Robotics Engineer and DevOps enthusiast. My mission is to create skill-learning platform that combats information overload by adhering to the set of principles: simplify, prioritize, and execute.

 <https://github.com/MichaelRobotics>



Ask Personal AI Document assistant to learn interactively (FASTER)!



Completly new to Linux and Networking?

Essential for this PDF is a thorough knowledge of networking. I highly recommend the HTB platform's networking module, which offers extensive information to help build a comprehensive understanding.

HTB - Your Cyber Performance Center

We provide a human-first platform creating and maintaining high performing cybersecurity individuals and organizations.

 <https://www.hackthebox.com/>



What is Kubernetes?

Kubernetes is an open-source platform that automates the deployment, scaling, and management of containerized applications. It helps manage clusters of nodes running containers, ensuring efficient and reliable operation.

How Kubernetes clusters are made?

Kubernetes clusters consist of a control plane and multiple worker nodes. The control plane manages cluster operations, while worker nodes run the actual container workloads.

Why and When use Kubernetes

Kubernetes is ideal for deploying scalable, resilient, and automated containerized applications. It is used when managing multiple containers across different environments is necessary.

Example: Running a microservices-based e-commerce platform that scales up during peak hours.

System Requirements

- RAM: 2 GB per node (1 GB can work for testing but may lead to limited performance)
- 10 GB free storage
- Ubuntu

Kubernetes: Main components & packages

- **kube-apiserver:** Central management component that exposes the Kubernetes API; acts as the front-end for the cluster.
- **etcd:** Distributed key-value store for storing all cluster data, ensuring data consistency across nodes.
- **kube-scheduler:** Assigns pods to available nodes based on resource requirements and policies.
- **kube-controller-manager:** Manages core controllers that handle various functions like node status, replication, and endpoints.
- **kubelet:** Agent that runs on each node, responsible for managing pods and their containers.
- **kube-proxy:** Manages networking on each node, ensuring communication between pods and services within the cluster.

Kubernetes EKS: Containerization intro

1) Microservices and Containerization

Download project: `git clone https://github.com/MichaelRobotics/Kubernetes.git`

Modern apps use microservices, dividing them into small, independent, interactive services.

This improves scalability, maintenance, and deployment. Before deploying to Kubernetes EKS, microservices must be containerized. Containerization packages the app and dependencies into a portable runtime, ensuring consistent execution.

2) README.md

A README file is information from developer how to deploy app. It ensures proper setup, preventing errors and misconfigurations. It contains essential details like:

- Dependencies: Required software and configurations.
- Build Instructions: Steps to containerize the service.
- Environment Variables: Key settings.
- Networking: Service interactions.
- Security: Best practices.

Ignoring the README may cause issues. If missing, ask the developer or follow containerization best practices.

- Use lightweight base images.
- Define efficient Dockerfiles.
- Manage dependencies well.
- Configure entry points and variables.
- Follow security best practices.

Kubernetes EKS: C# containers

1) Otel implementation, Accounting service

In .NET OTEL is integrated due to OpenTelemetry.AutoInstrumentation package. All configurations are setup through instrument.sh script and environment variables.

This package enables telemetry collection without requiring manual instrumentation in the application code. It supports tracing, metrics, and logging for various .NET components, including:

- ASP.NET Core
- gRPC
- Kafka (via Confluent.Kafka)
- Database interactions
- HTTP requests

Go to src/accounting/Accounting.csproj and check how OTEL package dependency is applied

```
</PackageReference>  
<PackageReference Include="Microsoft.Extensions.Logging" Version="9.0.1" />  
<PackageReference Include="Microsoft.VisualStudio.Azure.Containers.Tools.Targets" Version="1.21.0" />  
<PackageReference Include="OpenTelemetry.AutoInstrumentation" Version="1.9.0" />  
</ItemGroup>
```

2) Building .NET apps

On Linux, you need to install the .NET SDK to build and run .NET applications.

```
wget https://packages.microsoft.com/config/ubuntu/$(lsb_release -rs)/packages-microsoft-prod.deb -O packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
sudo apt update
sudo apt install -y dotnet-sdk-8.0
```

When building an already existing .NET project on Linux, you need to copy all necessary files and specify where the dependencies are located. Ensure that you have all required files, typically including:

```
.csproj or .sln (Project or Solution file)
Program.cs (or equivalent entry point)
appsettings.json (if applicable)
```

Run restore command pointing towards dependencies .csproj file or .sln file

```
dotnet restore "/path/to.csproj"
```

If you want to check for any compilation errors, you can separate the compilation process from the deployment stage. To do this, explicitly specify the path to the .csproj or .sln file during the compilation step.

```
dotnet build /path/to.csproj -c $<BUILD_CONFIGURATION_IF_HAVE_ANY>
```

This approach is particularly useful for application testing purposes. When using a tool like a Dockerfile, this stage should be cached to optimize the build process. As a result, even if you later run the publish command—which both compiles and deploys the application—all previously compiled files will be reused, reducing build time and improving efficiency.

```
dotnet publish /path/to.csproj -c $BUILD_CONFIGURATION
```

Finally, it's time to run the application. When working in a development environment, we typically use the dotnet run command. However, since we aim to deploy the application as it would be in a production environment, we should run it directly from the main .dll file instead.

```
dotnet /path/to/Program.dll
```

Now, let's check the README.md file provided by the microservice author, especially if the Dockerfile is missing.

3) Readme

Navigate towards README.md file:

Accounting Service

This service consumes new orders from a Kafka topic.

Local Build

To build the service binary, run:

```
cp pb/demo.proto src/accounting/proto/demo.proto # root context
dotnet build # accounting service context
```

Bump dependencies

To bump all dependencies run in Package manager:

```
Update-Package -ProjectName Accounting
```



The author mentions that for a local build, you need to copy the pb/demo/proto file and then run dotnet build in the accounting directory, presumably assuming that you are familiar with the process of building .NET applications and need only to know where the source code is located.

Docker Build

From the root directory, run:

```
docker compose build accounting
```



The project already includes existing Docker Compose files and Dockerfiles, and the author also provided instructions on how to easily run these files using Docker.

4) Accounting dockerfile

Finally lets check Dockerfile. I assume you are in Accounting directory:

```
cat Dockerfile
```

Authore Uses the .NET SDK image.

`${BUILDPLATFORM}`: Specifies the platform architecture for building the Docker image (e.g., linux/amd64, linux/arm64).

`TARGETARCH`: Specifies the target architecture for building the .NET project (e.g., x64, arm64).

`BUILD_CONFIGURATION`: Specifies the build configuration for the .NET project (e.g., Release, Debug).

```
FROM --platform=${BUILDPLATFORM} mcr.microsoft.com/dotnet/sdk:8.0 AS builder
ARG TARGETARCH
ARG BUILD_CONFIGURATION=Release
```

All files specified in README.md were copied into container

```
FROM --platform=${BUILDPLATFORM} mcr.microsoft.com/dotnet/sdk:8.0 AS builder
WORKDIR /src
COPY ["/src/accounting/", "Accounting/"]
COPY ["/pb/demo.proto", "Accounting/proto/"]
```

All necessary dependencies restored

```
RUN dotnet restore "./Accounting/Accounting.csproj" -r linux-$TARGETARCH
```

Developer separated compilation and deployment to catch any errors in compilation phase

```
WORKDIR "/src/Accounting"
```

```
RUN dotnet build "./Accounting.csproj" -r linux-$TARGETARCH -c $BUILD_CONFIGURATION -  
o /app/build
```

Then next step is to deploy application. Author used all data declared in previous stage

The `-p:UseAppHost=false` option in a .NET build command disables the creation of a platform-specific executable, we will execute from file with `.dll` extension.

```
FROM builder AS publish
```

```
ARG TARGETARCH
```

```
ARG BUILD_CONFIGURATION=Release
```

```
RUN dotnet publish "./Accounting.csproj" -r linux-$TARGETARCH -c $BUILD_CONFIGURATION  
-o /app/publish /p:UseAppHost=false
```

This final phase prepares the container for running the application in a production environment, taking only the necessary files from the build process.

```
FROM mcr.microsoft.com/dotnet/aspnet:8.0
```

```
USER app
```

```
WORKDIR /app
```

```
COPY --from=publish /app/publish .
```

creating a non-root user as the default user in a container is a security best practice. This minimizes the potential damage if the container is compromised, as attackers won't have root privileges to make system-wide changes.

```
USER root
RUN mkdir -p "/var/log/opentelemetry/dotnet"
RUN chown app "/var/log/opentelemetry/dotnet"
RUN chown app "/app/instrument.sh"
USER app
```

The instrument.sh file plays a crucial role in setting up the necessary configurations and environment variables for OpenTelemetry within the container

```
ENTRYPOINT ["/instrument.sh", "dotnet", "Accounting.dll"]
```

Kubernetes EKS: Java containers

1) Otel implementation, AD service explanation

AD service relies on the OpenTelemetry Java agent to automatically instrument common operations (e.g., gRPC calls, HTTP requests, database queries), and configures the OpenTelemetry SDK. The agent is passed into the process using the `-javaagent` command line argument.

2) Spans and metrics

A span represents a single unit of work or operation within a distributed system. Spans track the flow of requests across services, capturing execution time, errors, and dependencies between operations.

Metrics, on the other hand, provide aggregated insights over time. They measure key performance indicators such as request counts, response latencies, error rates, and system resource usage. While spans help debug individual requests, metrics reveal trends and overall system health.

Together, spans and metrics provide both detailed tracing (what happened in a single request?) and high-level monitoring (how is the system performing over time?).

3) Manual Instrumentation with Spans

While spans are created automatically, they often lack application-specific context. Attributes enrich spans with meaningful data:

```
Span span = Span.current(); // Get the current span
span.setAttribute("app.ads.contextKeys", req.getContextKeysList().toString());
span.setAttribute("app.ads.contextKeys.count", req.getContextKeysCount());
```

This example records how many and which context keys were used in an ad request.

Auto-instrumentation covers many standard operations, but critical custom logic (e.g., random ad selection) is not automatically traced. Creating spans allows you to track these steps:

```
Tracer tracer = GlobalOpenTelemetry.getTracer("adservice");
Span span = tracer.spanBuilder("getRandomAds").startSpan();
try (Scope ignored = span.makeCurrent()) {
    Collection<Ad> allAds = adsMap.values();
    for (int i = 0; i < MAX_ADS_TO_SERVE; i++) {
        ads.add(Iterables.get(allAds, random.nextInt(allAds.size())));
    }
    span.setAttribute("app.ads.count", ads.size());
} finally {
    span.end(); // End the span after operation
}
```

With a new span, you get nested spans that pinpoint where delays occur.

Events help capture key occurrences within a span, such as errors:

```
span.addEvent("Error", Attributes.of(AttributeKey.stringKey("exception.message"),
e.getMessage()));
span.setStatus(StatusCode.ERROR);
```

This keeps errors within the same trace, making debugging easier.

4) Manual Instrumentation with Metrics

Customization of metrics itself can be achieved with “Meter” instances.

Similar to creating spans, the first step in creating metrics is initializing a Meter instance, e.g.

```
GlobalOpenTelemetry.getMeter("adservice").
```

From there, use the various builder methods available on the Meter instance to create the desired metric instrument, e.g.:

```
meter
    .counterBuilder("app.ads.ad_requests")
    .setDescription("Counts ad requests by request and response type")
    .build();
```

\app.ads.ad_requests – A counter tracking ad requests, with dimensions indicating whether targeting was based on context keys and whether the response contained targeted or random ads.

Here is example from the Ad service code:

```
private static final AdService service = new AdService();
private static final Tracer tracer = GlobalOpenTelemetry.getTracer("ad");
private static final Meter meter = GlobalOpenTelemetry.getMeter("ad");

private static final LongCounter adRequestsCounter =
    meter
        .counterBuilder("app.ads.ad_requests")
        .setDescription("Counts ad requests by request and response type")
        .build();
```

5) Building Java apps with gradle

If you create app from scratch:

```
gradle init --type java-application
```

This will create directory structure:

```
MyJavaApp/
|  — build.gradle
|  — settings.gradle
|  — gradlew (Unix)
|  — gradlew.bat (Windows)
|  — gradle/
|  — src/
|    |  — main/
|    |    |  — java/com/example/App.java
|    |  — test/
|    |    |  — java/com/example/AppTest.java
|  — .gitignore
```

settings.gradle -> Defines the project name and includes subprojects if needed.

src -> Where all source code and tests will be located

gradlew & gradlew.bat -> gradlew (on Linux/macOS) and gradlew.bat (on Windows) are Gradle Wrapper scripts. They allow you to run Gradle without installing it manually on your system. Instead, they automatically download and use the specified Gradle version for the project.

gradle/ -> the gradle/ directory is created when you use the Gradle Wrapper (gradlew). It contains files necessary to run Gradle without requiring a system-wide Gradle installation.

build.gradle -> Most important from build perspective. This is the core build script that configures dependencies, plugins, and tasks. For example

```
plugins {  
    id 'application'  
    id 'java'  
}  
  
repositories {  
    mavenCentral()  
}  
  
dependencies {  
    implementation 'com.google.guava:guava:32.1.2-jre'  
    testImplementation 'org.junit.jupiter:junit-jupiter:5.9.1'  
}  
  
application {  
    mainClass = 'com.example.App'  
}  
  
tasks.named('test') {  
    useJUnitPlatform()  
}
```

Plugins extend Gradle. "application" enables tasks like run, "java" adds compileJava, test, and jar.

Repositories define where dependencies come from. "mavenCentral()" uses Maven Central, a large Java library repository.

Dependencies list required libraries. "implementation" adds production dependencies (e.g., Guava), "testImplementation" adds test-only dependencies (e.g., JUnit 5).

Application config sets the entry point. "mainClass" defines the main class (com.example.App) for execution.

Task runs tasks. It ensures JUnit 5 is used. "useJUnitPlatform()" runs tests with JUnit Jupiter.

If you have installed gradle you can follow standard procedure to build, deploy or even test:

compile project:

```
gradle build
```

Run the Java application with:

```
gradle run
```

If you have JUnit tests or other:

```
gradle test
```

package the application into a JAR file

```
gradle jar
```

run the JAR file in CLI to run locally or in docker container using k8s / docker itself

```
java -jar build/libs/MyJavaApp.jar
```

To upload a project to a repository (such as Maven or a custom repository) use:

```
gradle publish
```

You need to define the repository URL and credentials in build.gradle

To run Java apps with Gradle automatically and without manual installation, use gradlew

This is basic operations performed by gradlew in case of building an app:

- gradlew detects the build.gradle file in the current directory.
- It invokes Gradle Wrapper (gradle-wrapper.jar), which reads the build.gradle file.
- Gradle uses the information in build.gradle to: resolve dependencies, compile source code, run tests, package the application

6) README.md

Navigate towards README.md file:

Building Locally

The Ad service requires at least JDK 17 to build and uses gradlew to compile/install/distribute. Gradle wrapper is already part of the source code. To build Ad Service, run:

```
./gradlew installDist
```

It will create an executable script `src/ad/build/install/oteldemo/bin/Ad`.

To run the Ad Service:

```
export AD_PORT=8080
export FEATURE_FLAG_GRPC_SERVICE_ADDR=featureflagservice:50053
./build/install/opentelemetry-demo-ad/bin/Ad
```

Ad service requires at least JDK 17 to build and uses Gradle Wrapper (gradlew) to compile, install, and distribute. The Gradle wrapper is already included in the source code.

To build the Ad Service, run: `./gradlew installDist`

This will create an executable script at:

`src/ad/build/install/oteldemo/bin/Ad`

To run the Ad service, you need to set environment variables:

```
export AD_PORT=8080
export FEATURE_FLAG_GRPC_SERVICE_ADDR=featureflagservice:50053
```

Then start the service:

```
./build/install/opentelemetry-demo-ad/bin/Ad
```

Developer provided Dockerfile already, so you can just use docker tool:

Building Docker

From the root of `opentelemetry-demo`, run:

```
docker build --file ./src/ad/Dockerfile ./
```



7) Ad Dockerfile

Use Eclipse Temurin JDK 21 as the base image for compiling Java code. As BUILDPLATFORM you can probably use linux/amd64.

```
FROM --platform=${BUILDPLATFORM} eclipse-temurin:21-jdk AS builder
```

Define WORKDIR as `/usr/src/app/` where the source code and build artifacts will be stored. ARG will be later used to pass opentelemetry data into container image.

```
ARG _JAVA_OPTIONS
WORKDIR /usr/src/app/
```

Copy Gradle Wrapper scripts (gradlew), build configuration (build.gradle), and settings (settings.gradle) to the working directory.

Also copy the gradle/ directory, which contains local Gradle configurations.

```
COPY ./src/ad/gradlew* ./src/ad/settings.gradle* ./src/ad/build.gradle ./
COPY ./src/ad/gradle ./gradle
```

Grants execution permission to gradlew.

Run ./gradlew, ensuring Gradle is set up and dependencies are cached via downloadRepos (which downloads external libraries).

```
RUN chmod +x ./gradlew
RUN ./gradlew
RUN ./gradlew downloadRepos
```

Copy Java source files (src/ad/) and Protocol Buffers (pb/) to the build directory.

```
COPY ./src/ad/ ./
COPY ./pb/ ./proto
```

Grant execution permission to gradlew again (in case of file permission issues).

Build and install the application using installDist, which creates an executable distribution under build/install/.

```
RUN chmod +x ./gradlew
RUN ./gradlew installDist -PprotoSourceDir=./proto
```

Proto files, or Protocol Buffers files, are used to define the structure of the data you want to serialize and communicate between systems or services. Protocol Buffers (often referred to as Protobuf) is a method developed by Google for serializing structured data, similar to JSON

Use a smaller JRE 21 runtime (instead of JDK) since compilation is already done.
This reduces the image size and improves performance.

```
FROM eclipse-temurin:21-jre
```

Define working directory again.

Use build-time arguments `OTEL_JAVA_AGENT_VERSION` and `_JAVA_OPTIONS` for instrumentation and JVM options.

```
ARG OTEL_JAVA_AGENT_VERSION
ARG _JAVA_OPTIONS
WORKDIR /usr/src/app/
```

Copy the built application from the previous stage (builder).

```
COPY --from=builder /usr/src/app/ ./
```

Download OpenTelemetry agent (used for tracing, metrics, and monitoring).
Set `JAVA_TOOL_OPTIONS` to attach the OpenTelemetry Java agent when running the app.

```
ADD --chmod=644 https://github.com/[...]
ENV JAVA_TOOL_OPTIONS=-javaagent:/usr/src/app/opentelemetry-javaagent.jar
```

Expose the Ad Service's port (`AD_PORT`).

Define the startup command, run the installed Ad binary.

```
EXPOSE ${AD_PORT}
ENTRYPOINT [ "./build/install/opentelemetry-demo-ad/bin/Ad" ]
```

Kubernetes EKS: PHP containers

1) Otel implementation, Quote service explanation

Quote service is responsible for calculating shipping costs, based on the number of items to be shipped. The quote service is called from Shipping Service via HTTP.

The Quote Service is implemented using the Slim framework and php-di for managing the Dependency Injection.

The PHP instrumentation may vary when using a different framework.

In this demo, the OpenTelemetry SDK has been automatically created as part of SDK autoloading, which happens as part of composer autoloading. Autoloading in PHP is a mechanism that automatically loads class files when they are needed

This is enabled by setting the environment variable `OTEL_PHP_AUTOLOAD_ENABLED=true`.
autoload initialization is handled in `index.php`:

```
require __DIR__ . '/../vendor/autoload.php';
```

2) Manual instrumentation with spans

In PHP there are multiple ways to create or obtain a Tracer, in this example we obtain one from the global tracer provider which was initialized above, as part of SDK autoloading:

```
$tracer = Globals::tracerProvider()->getTracer('manual-instrumentation');
```

Creating a span manually can be done via a Tracer. The span will child of the active span in the current execution context:

```
$span = Globals::tracerProvider()
  ->getTracer('manual-instrumentation')
  ->spanBuilder('calculate-quote')
  ->setSpanKind(SpanKind::KIND_INTERNAL)
  ->startSpan();
/* calculate quote */
$span->end();
```

Adding attributes to a span is accomplished using `setAttribute` on the span object. In the `calculateQuote` function 2 attributes are added to the `childSpan`.

```
$childSpan->setAttribute('app.quote.items.count', $numberOfItems);
$childSpan->setAttribute('app.quote.cost.total', $quote);
```

Adding span events is accomplished using `addEvent` on the span object. In the `getquote` route span events are added. Some events have additional attributes, others do not.

Adding a span event without attributes:

```
$span->addEvent('Received get quote request, processing it');
```

Adding a span event with additional attributes:

```
$span->addEvent('Quote processed, response sent back', [
  'app.quote.cost.total' => $payload
]);
```

3) Manual instrumentation with metrics

In this demo, metrics are emitted by the batch trace and logs processors. The metrics describe the internal state of the processor, such as number of exported spans or logs, the queue limit, and queue usage.

You can enable metrics by setting the environment variable

OTEL_PHP_INTERNAL_METRICS_ENABLED to true.

A manual metric is also emitted, which counts the number of quotes generated, including an attribute for the number of items.

A counter is created from the globally configured Meter Provider, and is incremented each time a quote is generated:

```
static $counter;  
$counter ??= Globals::meterProvider()  
    ->getMeter('quotes')  
    ->createCounter('quotes', 'quotes', 'number of quotes calculated');  
$counter->add(1, ['number_of_items' => $numberOfItems]);
```

Metrics accumulate and are exported periodically based on the value configured in OTEL_METRIC_EXPORT_INTERVAL.

4) Build in PHP with composer

Composer is a tool for dependency management in PHP. It also handles autoloading and other essential project configurations. To create PHP project template with composer:

```
composer init
```

This command will walk create a composer.json file, which defines your project and any dependencies you will use. For example, check Quote service composer.json:

```
{
  "name": "opentelemetry-demo/quote",
  "description": "Quote Service part of OpenTelemetry Demo",
  "license": "Apache-2.0",
  "require": {
    "php": ">= 8.3",
    "ext-json": "*",
    "ext-pcntl": "*",
    "monolog/monolog": "3.8.1",
    "open-telemetry/api": "1.2.2",
  },
  [..]
  "open-telemetry/opentelemetry-auto-slim": "1.0.7",
  "php-di/php-di": "7.0.8",
  "php-di/slim-bridge": "3.4.1",
  "php-http/guzzle7-adapter": "1.1.0",
  [..]
  "config": {
    "allow-plugins": {
      "php-http/discovery": false
    }
  }
}
```

Project Info (name, description, license)

Dependencies (require) – PHP extensions and libraries like monolog/monolog for logging and open-telemetry/sdk for observability.

Configuration (config) – Composer behavior, e.g., plugin permissions.

To download all specified dependencies in composer.json and setup rest of important files:

```
composer install
```

It will create project files, where src/ public/ config/ views/ folders you need to add yourself.

```
/your-project
| ——— composer.json    # Defines project dependencies & metadata
| ——— composer.lock    # Locks exact package versions for consistency
| ——— vendor/          # Installed dependencies
| ——— src/              # Application source code
| ——— public/           # Public-facing files
|   |——— index.php      # Main entry point for the application
| ——— config/           # Configuration files
| ——— views/            # Template files (if applicable)
```

after writing all code, you can optimize autoloading:

```
composer dump-autoload --optimize
```

and ran some tests for example with PHPUnit

```
php vendor/bin/phpunit tests/
```

In laravel use artisan:

```
php vendor/bin/phpunit tests/
```

5) Readme

Navigate towards README.md file:

Development

To build and run the quote service locally:

```
docker build src/quote --target base -t quote
cd src/quote
docker run --rm -it -v $(pwd):/var/www -e QUOTE_PORT=8999 -p "8999:8999" quote
```

Then, send some curl requests:

```
curl --location 'http://localhost:8999/getquote' \
--header 'Content-Type: application/json' \
--data '{"numberOfItems":3}'
```

Unfortunately, author did not provide how to created Dockerfile, but we can deduce that app runs on port 8999.

To test curl on /getquote endpoint.

Docker Build

To build the quote service, run the following from root directory of opentelemetry-demo

```
docker compose build quote
```

Run the service

Execute the below command to run the service.

```
docker compose up quote
```

If you want most simple way of deploying this service, use docker compose.

6) Quote Dockerfile

Use the PHP 8.3 CLI image (php:8.3-cli) as the base for the first stage. PHP 8.3 and provide a basic environment for running PHP scripts from the command line.

```
FROM php:8.3-cli AS base
```

Download install-php-extensions script, make it executable and install PHP extensions: opcache for caching, pcntl for process control, protobuf for data serialization, and opentelemetry for monitoring and tracing.

```
ADD https://github.com/mlocati/docker-php-extension-  
installer/releases/latest/download/install-php-extensions /usr/local/bin/  
RUN chmod +x /usr/local/bin/install-php-extensions \  
&& install-php-extensions \  
    opcache \  
    pcntl \  
    protobuf \  
    opentelemetry
```

Set the working directory to /var/www. Run php public/index.php when the container starts. Switch user to www-data for security. Expose the port defined by QUOTE_PORT (8999).

```
WORKDIR /var/www  
CMD ["php", "public/index.php"]  
USER www-data  
EXPOSE ${QUOTE_PORT}
```

Use the official Composer 2.7 image to install dependencies. Set the working directory to /tmp/ for the Composer install process. Copy composer.json from the local directory to /tmp/ inside the container.

```
FROM composer:2.7 AS vendor
WORKDIR /tmp/
COPY ./src/quote/composer.json .
```

Use the official Composer 2.7 image to install dependencies. Set the working directory to /tmp/ for the Composer install process. Copy composer.json from the local directory to /tmp/ inside the container. Install dependencies with composer install, using the following flags:

```
--ignore-platform-reqs: Ignores platform requirements.
--no-interaction: Prevents user input prompts.
--no-plugins: Disables Composer plugins.
--no-scripts: Skips any scripts in composer.json.
--no-dev: Excludes development dependencies.
--prefer-dist: Prefers .tar/.zip distributions for faster installation.

RUN composer install \
    --ignore-platform-reqs \
    --no-interaction \
    --no-plugins \
    --no-scripts \
    --no-dev \
    --prefer-dist
```

Start the final stage of the build, using the base image from the first stage (PHP 8.3 CLI with installed extensions). Copy the vendor/ directory, containing Composer dependencies, from the vendor stage into /var/www/ in the final image. Copy the application source code from ./src/quote/ on the host to /var/www in the container.

```
FROM base AS final
```

```
COPY --from=vendor /tmp/vendor/ ./vendor/
```

```
COPY ./src/quote/ /var/www
```

Kubernetes EKS: Ruby containers

1) Otel implementation, Email service explanation

Email service will send a confirmation email to the user when an order is placed.

You will need to require the core OpenTelemetry SDK and exporter Ruby gems, as well as any gem that will be needed for auto-instrumentation libraries (ie: Sinatra)

```
require "opentelemetry/sdk"  
require "opentelemetry/exporter/otlp"  
require "opentelemetry/instrumentation/sinatra"
```

The Ruby SDK uses OpenTelemetry standard environment variables to configure OTLP export, resource attributes, and service name automatically. When initializing the OpenTelemetry SDK, you will also specify which auto-instrumentation libraries to leverage (ie: Sinatra)

```
OpenTelemetry::SDK.configure do |c|  
  c.use "OpenTelemetry::Instrumentation::Sinatra"  
end
```

2) Manual instrumentation using spans

Within the execution of auto-instrumented code, you can get the current span from the context:

```
current_span = OpenTelemetry::Trace.current_span
```

Adding multiple attributes to a span is accomplished using `add_attributes` on the span object:

```
current_span.add_attributes({  
  "app.order.id" => data.order.order_id,  
})
```

Adding a single attribute can be done using `set_attribute`:

```
span.set_attribute("app.email.recipient", data.email)
```

New spans can be created and placed into active context using `in_span` from an OpenTelemetry Tracer object. When used in conjunction with a `do..end` block, the span will automatically end when the block finishes execution:

```
tracer = OpenTelemetry.tracer_provider.tracer('emailservice')  
  
tracer.in_span("send_email") do |span|  
  # logic in context of span here  
end
```


Span events allow adding timestamped annotations to a span to provide more context about what happens during its execution. You can add an event to a span using `add_event`

```
span.add_event("email_queued", attributes: { "queue.name" => "high_priority" })
```

3) Manual instrumentation using metrics

To set up metrics, first get a meter from the Meter Provider:

```
meter = OpenTelemetry.meter_provider.meter("my_service")
```

Then create a metric. For example, counter metric. A counter is useful for tracking the number of occurrences of an event (e.g., requests received).

```
request_counter = meter.create_counter("http.requests", description: "Total HTTP requests")
request_counter.add(1, attributes: { "http.method" => "GET", "http.route" => "/home" })
```

4) Build in Ruby with gemfile

At first familiarize yourself with `gem`. A gem in Ruby is a package or library that provides reusable code to extend or enhance the functionality of your Ruby application.

To initialize Ruby project, install `bundle`, assuming you have already installed `ruby`

```
gem install bundler
```

And init project

```
mkdir my_ruby_app && cd my_ruby_app
bundle init # Generates a Gemfile
```

Project folder will contain Gemfile, which like in PHP with composer composer.json is accountable for dependency management

```
/my_ruby_app
├── Gemfile # Defines project dependencies (e.g., 'rspec', 'rails')
```

Example Gemfile may look like this:

```
source "https://rubygems.org"

gem "sinatra" # Example for a web application
gem "rack"    # For middleware support
```

Install dependencies

```
bundle install
```

Now your directory structure will look like this:

```
/my_ruby_app
├── Gemfile          # Your project's gem dependencies
├── Gemfile.lock     # Locks the gem versions for consistency
├── /vendor          # default where gems are installed
├── /bin             # default dir for executable scripts
├── /bundler.d       # Bundler configuration (if present)
├── /spec            # Where your test files go (if using RSpec or other testing frameworks)
├── .rspec           # RSpec configuration (if using RSpec)
└── /other_project_files # Other files in your project
```

Now create app entry in directory and write there some code:

```
touch app.rb
```

```
require 'sinatra'
```

```
get '/' do
  "Hello, Ruby App!"
end
```

run code:

```
bundle exec ruby app.rb
```

you can create executable by specifying path to ruby and yours project library:

```
#!/usr/bin/env ruby
require_relative '../lib/my_ruby_app'

puts "Running My Ruby App!"
```

or even run tests. Downloading tests gem

```
gem 'rspec'
```

Create tests directory and write there some tests:

```
rspec --init #test path is in spec/app_spec.rb):
rspec #run tests
```

You can build your own gem. To do it, create folder structure for your gem:

```
bundle gem my_ruby_app
```

The gemspec file defines the metadata and the files to include in the gem.

```
Gem::Specification.new do |spec|  
  spec.name      = "my_ruby_app"  
  spec.version   = "0.1.0"  
  spec.authors  = ["Your Name"]  
  spec.summary   = "A Ruby app"  
  spec.files    = Dir["lib/**/*.rb"]  
  spec.require_paths = ["lib"]  
  
end
```

create gem

```
gem build my_ruby_app.gemspec
```

install locally:

```
gem install ./my_ruby_app-0.1.0.gem
```

5) Readme

Email Service

The Email service "sends" an email to the customer with their order details by rendering it as a log message. It expects a JSON payload like:

```
{  
  "email": "some.address@website.com",  
  "order": "<serialized order protobuf>"  
}
```



Local Build

We use `bundler` to manage dependencies. To get started, simply `bundle install`.

There is actually not that much needed to create Dockerfile. Copy project and install all dependencies.

Running locally

You may run this service locally with `bundle exec ruby email_server.rb`.

then exec main program entry.

6) Email Dockerfile

Set up Ruby 3.2.2 on a minimal Debian-based image. A slim image keeps the container small and efficient

FROM ruby:3.2.2-slim AS base

Set /tmp as the working directory, copy Gemfile and Gemfile.lock to allow dependency caching, reducing the final image size and speeding up future builds.

```
FROM base AS builder
WORKDIR /tmp
COPY ./src/email/Gemfile ./src/email/Gemfile.lock ./
```

update packages, install tools like make and gcc, install Ruby gems with native dependencies.

```
RUN apt-get update && apt-get install build-essential -y && bundle install
```

Create a clean image to run the app, set the working directory to /email_server, copy the source code, and adjust permissions for Gemfile.lock to ensure proper app functionality.

```
FROM base AS release
WORKDIR /email_server
COPY ./src/email/ .
RUN chmod 666 ./Gemfile.lock
```

Copy installed gems from the builder, expose service port, define the startup command to run the email service.

```
COPY --from=builder /usr/local/bundle/ /usr/local/bundle/
EXPOSE ${EMAIL_PORT}
ENTRYPOINT ["bundle", "exec", "ruby", "email_server.rb"]
```

Kubernetes EKS: C++ containers

1) Otel implementation, Currency service explanation

Currency service provides functionality to convert amounts between different currencies.

The OpenTelemetry SDK is initialized from main using the initTracer function defined in tracer_common.h

```
void initTracer(){  
[...]  
}
```

Lets Understand this function line by line.

Code creates an OTLP gRPC exporter using OpenTelemetry's factory method.

Exporter is responsible for sending traces (spans)

```
auto exporter = opentelemetry::exporter::otlp::OtlpGrpcExporterFactory::Create();
```

The SimpleSpanProcessor immediately exports each span as it's finished.

```
auto processor =  
    opentelemetry::sdk::trace::SimpleSpanProcessorFactory::Create(std::move(exporter));
```

A vector is created to store span processors, and the SimpleSpanProcessor is moved into it.

```
std::vector<std::unique_ptr<opentelemetry::sdk::trace::SpanProcessor>> processors;  
processors.push_back(std::move(processor));
```

A TracerContext is created with the span processors and manages the tracing pipeline's lifecycle.

```
std::shared_ptr<opentelemetry::sdk::trace::TracerContext> context =  
    opentelemetry::sdk::trace::TracerContextFactory::Create(std::move(processors));
```

A TracerProvider is created as a factory for tracers, which generate spans representing application operations.

```
std::shared_ptr<opentelemetry::trace::TracerProvider> provider =  
    opentelemetry::sdk::trace::TracerProviderFactory::Create(context);
```

The global TracerProvider is set, enabling components to automatically use it for trace generation without explicit configuration.

```
opentelemetry::trace::Provider::SetTracerProvider(provider);
```

HttpTraceContext is set as the global propagator, responsible for injecting and extracting trace context across service boundaries to ensure correct distributed tracing.

```
opentelemetry::context::propagation::GlobalTextMapPropagator::SetGlobalPropagator(  
    opentelemetry::nostd::shared_ptr<opentelemetry::context::propagation::TextMapPropagator>(  
        new opentelemetry::trace::propagation::HttpTraceContext()));
```


2) Manual instrumentation using spans

New spans can be created and started using `Tracer->StartSpan("spanName", attributes, options)`. After a span is created you need to start and put it into active context using `Tracer->WithActiveSpan(span)`.

```
std::string span_name = "CurrencyService/Convert";
auto span =
    get_tracer("currencyservice")->StartSpan(span_name,
        {{SemanticConventions::kRpcSystem, "grpc"},
         {SemanticConventions::kRpcService, "oteldemo.CurrencyService"},
         {SemanticConventions::kRpcMethod, "Convert"},
         {SemanticConventions::kRpcGrpcStatusCode, 0}},
        options);
auto scope = get_tracer("currencyservice")->WithActiveSpan(span);
```

You can add an attribute to a span using `Span->SetAttribute(key, value)`.

```
span->SetAttribute("app.currency.conversion.from", from_code);
span->SetAttribute("app.currency.conversion.to", to_code);
```

Adding span events is accomplished using `Span->AddEvent(name)`.

```
span->AddEvent("Conversion successful, response sent back");
```

In distributed tracing systems, context propagation refers to passing along trace context (e.g., trace ID, span ID) as requests flow between services or components. In C++, context propagation is not automatically handled. `GrpcServerCarrier` is the class used to handle context propagation in the gRPC

GrpcServerCarrier handle context propagation by extracting and injecting trace context into gRPC requests. It implements the TextMapCarrier interface from OpenTelemetry, which provides methods to get and set trace context in the form of key-value pairs.

```
class GrpcServerCarrier : public opentelemetry::context::propagation::TextMapCarrier
{
public:
    GrpcServerCarrier(ServerContext *context) : context_(context) {}
    GrpcServerCarrier() = default;

    virtual opentelemetry::nostd::string_view Get(
        opentelemetry::nostd::string_view key) const noexcept override
    {
        auto it = context_>client_metadata().find(key.data());
        if (it != context_>client_metadata().end())
        {
            return it->second.data();
        }
        return "";
    }

    virtual void Set(opentelemetry::nostd::string_view key,
        opentelemetry::nostd::string_view value) noexcept override
    {
        // Not required for server
    }

    ServerContext *context_;
};
```

3) Manual instrumentation using metrics

The `initMeter()` function is responsible for setting up the `MeterProvider` and exporting the metrics

```
void initMeter()
{
    // Build MetricExporter
    otlp_exporter::OtlpGrpcMetricExporterOptions otlpOptions;
    auto exporter = otlp_exporter::OtlpGrpcMetricExporterFactory::Create(otlpOptions);

    // Build MeterProvider and Reader
    metric_sdk::PeriodicExportingMetricReaderOptions options;
    std::unique_ptr<metric_sdk::MetricReader> reader{
        new metric_sdk::PeriodicExportingMetricReader(std::move(exporter), options) };

    auto provider = std::shared_ptr<metrics_api::MeterProvider>(new metric_sdk::MeterProvider());
    auto p = std::static_pointer_cast<metric_sdk::MeterProvider>(provider);

    // Add MetricReader to MeterProvider
    p->AddMetricReader(std::move(reader));

    // Set the global MeterProvider
    metrics_api::Provider::SetMeterProvider(provider);
}
```

4) Build c++ containers with cmake

You need to create c++ project directory yourself. Usual structure looks like this:

```
my_project/  
| — CMakeLists.txt  
| — src/  
|   | — main.cpp  
| — build/
```

CMakeLists.txt file defines all needed dependencies, its structure looks like this:

```
cmake_minimum_required(VERSION 3.10) # Set the minimum required version of CMake  
  
project(MyProject VERSION 1.0) # Define the project name and version  
  
set(CMAKE_CXX_STANDARD 17 ) # Specify C++ standard  
  
add_executable(MyExecutable main.cpp src/file1.cpp src/file2.cpp) # Add source files  
  
include_directories(include) # Include directories (for headers)  
  
target_link_libraries(MyExecutable PRIVATE SomeLibrary) # Link libraries  
  
install(TARGETS MyExecutable DESTINATION bin) # Define installation rules  
  
enable_testing() # Enable testing (for unit tests)
```

First you need to install all necessary tools and compilers

```
apk update && apk add git cmake make g++ linux-headers
```

To let cmake generate all makefiles from CMakeList.txt run:

```
cmake ..
```

Then run make. It compiles source files into object files, links them into an executable or library, and stores the binaries in the build directory, following the Makefile generated by cmake .. or written manually.

```
make
```

or make install if you want to build and install compiled application to system:

```
make install
```

After running the above commands, the executable my_project will be installed to the directory you specified in the install() call (in this case, the bin/ directory).

If the install path was /usr/local/bin/, you could run the project from anywhere like so:

```
my_project
```

5) Readme

Unfortunately Readme only tells us, that app is listening on 7000 port.

Building docker image

To build the currency service, run the following from root directory of opentelemetry-demo

```
docker-compose build currency
```



Run the service

Execute the below command to run the service.

```
docker-compose up currency
```



Run the client

currencyclient is a sample client which sends some request to currency service. To run the client, execute the below command.

```
docker exec -it <container_name> currencyclient 7000
```



'7000' is port where currency listens to.

6) Currency Dockerfile

Use Alpine Linux 3.18 as the base image for building.

```
FROM alpine:3.18 AS builder
```

Update the package index and install build tools: git (cloning), cmake, make, g++ (compiler for C++ build), grpc-dev, protobuf-dev (gRPC), and linux-headers (system headers).

```
RUN apk update && apk add git cmake make g++ grpc-dev protobuf-dev linux-headers
```

Clone and build OpenTelemetry C++ with cmake (C++17, Release mode, OTLP gRPC, Abseil), compile source code using all CPU cores, and install the SDK.

```
RUN git clone --depth 1 --branch v${OPENTELEMETRY_CPP_VERSION}
https://github.com/open-telemetry/opentelemetry-cpp \
    && cd opentelemetry-cpp/ \
    && mkdir build \
    && cd build \
    && cmake .. -DCMAKE_CXX_STANDARD=17 -DCMAKE_POSITION_INDEPENDENT_CODE=ON \
        -DCMAKE_BUILD_TYPE=Release -DBUILD_TESTING=OFF \
        -DWITH_EXAMPLES=OFF -DWITH_OTLP_GRPC=ON -DWITH_ABSEIL=ON \
    && make -j$(nproc || sysctl -n hw.ncpu || echo 1) install && cd ../..
```

Copy the application source (currency) and protobuf file (demo.proto) for gRPC stub generation into the container.

```
COPY ./src/currency /currency
COPY ./pb/demo.proto /currency/proto/demo.proto
```

Navigate to the app directory, create and enter build/, run cmake to configure, then compile with use of all cpu cores and install the app.

```
RUN cd /currency \
    && mkdir -p build && cd build \
    && cmake .. \
    && make -j$(nproc || sysctl -n hw.ncpu || echo 1) install
```

Use a lightweight Alpine 3.18 image, install runtime dependencies (grpc-dev, protobuf-dev), and exclude build tools to minimize image size, then copy built files.

```
FROM alpine:3.18 AS release
```

```
RUN apk update && apk add grpc-dev protobuf-dev
```

```
COPY --from=builder /usr/local /usr/local
```

Expose the currency service port and set the entry point to run the currency service binary

```
EXPOSE ${CURRENCY_PORT}
```

```
ENTRYPOINT ["sh", "-c", "./usr/local/bin/currency ${CURRENCY_PORT}"]
```


Kubernetes EKS: Rust containers

1) Otel implementation, Shipping service explanation

This service is responsible for providing shipping information including pricing and tracking information, when requested from Checkout Service. Shipping service is built primarily with Tonic, Reqwest, and OpenTelemetry Libraries/Components.

Example OpenTelemetry initialization:

```
use opentelemetry::{global, sdk::{trace as sdktrace, resource::Resource}};
use opentelemetry_otlp::new_pipeline;
use tracing_subscriber::{Registry, layer::SubscriberExt};
use tracing_opentelemetry::OpenTelemetryLayer;
use std::env;

fn init_tracer() -> Result<sdktrace::Tracer, opentelemetry::trace::TraceError> {

global::set_text_map_propagator(opentelemetry::propagation::TraceContextPropagator::new());

    new_pipeline()
        .tracing()
        .with_exporter(opentelemetry_otlp::new_exporter().tonic().with_endpoint(
            env::var("OTEL_EXPORTER_OTLP_TRACES_ENDPOINT")
                .unwrap_or_else(|_| "http://otelcol:4317".to_string()),
        ))
        .with_trace_config(sdktrace::config().with_resource(Resource::new(vec![
            ("service.name", "my-rust-service"),
        ])))
        .install_batch(opentelemetry::runtime::Tokio)
    }

fn setup_tracing() {
    let telemetry_layer = OpenTelemetryLayer::new(init_tracer().expect("Failed to init tracer"));
    tracing::subscriber::set_global_default(Registry::default().with(telemetry_layer))
        .expect("Failed to set global tracing subscriber");
}
```

2) Spans

To apply automatical instrumentation, add `#instrument` to function:

```
#[instrument]
async fn my_grpc_handler() {
  // Automatically instruments the function
}
```

If you want to manually create a span:

```
let span = span!(Level::INFO, "custom_trace");
let _enter = span.enter();
```

You can add attributes and events to span:

```
use opentelemetry::{global, trace::{Tracer, Span}};
use tracing::{info, instrument};
use tracing_opentelemetry::OpenTelemetrySpanExt;

#[instrument] // Creates a span for the function
async fn my_function() {
  let tracer = global::tracer("my-tracer");
  let span = tracer.start("my_span");

  // # Add attributes
  span.set_attribute("user_id", 12345);
  span.set_attribute("operation", "db_query");

  // # Add events
  span.add_event("Started query".to_string(), vec![]);

  info!("Function complete");
}
```

3) Trace propagation

When building microservices, requests often travel through multiple services (e.g., Client → Service A → Service B → Database). Without proper trace propagation, each service creates separate traces, making it impossible to track a request across the system. To setup propagation, you setup interceptor which injects trace context into gRPC header and extractor on server side, which consumes data. In this example we used gRPC but it can be any other protocol, process works same way.

Client Side

```
use tonic::{transport::Channel, Request};
use tonic::codegen::InterceptedService;
use tracing::{info, instrument, Level};
use tracing_opentelemetry::OpenTelemetrySpanExt;

#[derive(Clone)]
pub struct GrpcInterceptor;

impl tonic::service::Interceptor for GrpcInterceptor {
    fn call(&mut self, mut req: Request<()>) -> Result<Request<()>, tonic::Status> {
        let span = tracing::span!(Level::INFO, "grpc_client_request", uri = %req.uri());
        global::get_text_map_propagator(|p| p.inject_context(&span.context(), req.metadata_mut()));
        info!("Sending gRPC request");
        Ok(req)
    }
}

#[instrument] // Creates a tracing span for this function
async fn grpc_client() -> Result<(), Box<dyn std::error::Error>> {
    Channel::from_static("http://[::1]:50051")
        .intercept(GrpcInterceptor)
        .connect()
        .await?;
    info!("Connected to gRPC server");
    Ok(())
}
```

Server Side

```
use tonic::{Request, Response, Status};
use tracing::{info, instrument, Level};
use tracing_opentelemetry::OpenTelemetrySpanExt;

pub mod my_grpc_service {
    tonic::include_proto!("my_grpc_service");
}

use my_grpc_service::my_grpc_service_server::{MyGrpcService, MyGrpcServiceServer};
use my_grpc_service::{MyRequest, MyResponse};

#[derive(Default)]
pub struct MyService;

#[tonic::async_trait]
impl MyGrpcService for MyService {
    #[instrument] // Creates a tracing span for the request
    async fn my_method(
        &self,
        request: Request<MyRequest>,
    ) -> Result<Response<MyResponse>, Status> {
        let span = tracing::span!(Level::INFO, "grpc_server_request", uri = %request.uri());
        global::get_text_map_propagator(|p| p.extract(&request.metadata())); // Extract trace
context
        info!("Processing request");
        Ok(Response::new(MyResponse { message: "Hello".into() }))
    }
}
```

4) Metrics

To retrieve metrics, first initialize a Prometheus exporter for OpenTelemetry metrics and set it as the global meter provider, exposing metrics on 0.0.0.0:9090

```
use opentelemetry_prometheus::exporter;
use opentelemetry::global;

fn init_metrics() {
    let provider = exporter::new_pipeline().with_http_server("0.0.0.0:9090").build().unwrap();
    global::set_meter_provider(provider); // # Initialize and set global meter provider
}
```

Define and initialize metrics for a gRPC server by creating a counter for tracking total requests and a histogram for measuring request latency using OpenTelemetry.

```
use opentelemetry::metrics::{Meter, Counter, Histogram};

fn init_grpc_metrics(meter: &Meter) -> (Counter<u64>, Histogram<f64>) {
    let request_counter = meter.u64_counter("grpc_requests_total").init(); // # Create counter for gRPC requests
    let latency_histogram = meter.f64_histogram("grpc_request_duration_seconds").init(); // # Create histogram for latency
    (request_counter, latency_histogram)
}
```

Below is server code. Track the number of gRPC requests using a counter and records the request latency with a histogram.

```
use tonic::{Request, Response, Status};
use std::time::Instant;

pub struct MyServiceImpl {
    request_counter: Counter<u64>, // # Counter for tracking requests
    latency_histogram: Histogram<f64>, // # Histogram for tracking request latency
}

#[tonic::async_trait]
impl MyService for MyServiceImpl {
    async fn my_method(&self, _: Request<>) -> Result<Response<>>, Status> {
        let start = Instant::now();
        self.request_counter.add(1, &[]); // # Increment request counter
        let _ = do_some_work().await; // Simulate work
        self.latency_histogram.record(start.elapsed().as_secs_f64(), &[]); // # Record request latency
        Ok(Response::new(()))
    }
}

async fn do_some_work() -> String { "done".to_string() }
```

initialize Prometheus metrics, set up a gRPC server with a custom service implementation, and start the server to listen on 0.0.0.0:50051.

```
use tonic::transport::Server;

fn main() {
    init_metrics(); // # Initialize Prometheus metrics

    let meter = global::meter("my-rust-service");
    let (request_counter, latency_histogram) = init_grpc_metrics(&meter);

    let service = MyServiceImpl { request_counter, latency_histogram };
    let addr = "0.0.0.0:50051".parse().unwrap();
    Server::builder()
        .add_service(MyServiceServer::new(service))
        .serve(addr)
        .await
        .unwrap();
}
```

Add exporter to prometheus yml:

```
scrape_configs:
  - job_name: "grpc_service"
    static_configs:
      - targets: ["localhost:9090"] # # Expose metrics to Prometheus at /metrics
```

5) Build Rust apps with Cargo

In Rust, the standard way to build and manage projects is using Cargo, the official package manager and build system. After rust is installed, create project. Dir structure contains main file and cargo.toml:

```
cargo new my_project
cd my_project
```

```
my_project/
|—— Cargo.toml
|—— src/
|   |—— main.rs (or lib.rs for libraries)
```

Cargo.toml file serves as the manifest for the Cargo package manager. It defines metadata, dependencies, build settings, and other configurations

```
[package]
name = "my_project"
version = "0.1.0"
edition = "2021"
license = "MIT"

[dependencies]
serde = "1.0"
tokio = { version = "1", features = ["full"] }

[dev-dependencies]
criterion = "0.5"

[build-dependencies]
cc = "1.0"

[features]
default = ["serde"]
```

To create binaries:

```
cargo build
```

The output binary is placed in target/debug/

For release, create optimized build without debugging features:

```
cargo build --release
```

Eventually you can build binaries and run project in one command for debug and release option:

```
cargo run
```

```
cargo run --release
```

You can check syntax errors before building binaries:

```
cargo check
```

Or run test using built in framework. Tests are written using `#[test]` attributes in `tests/` or `src/`.

```
cargo test
```

To formatting or linting code:

```
cargo fmt
```

```
cargo clippy
```


6) Cross-compilation feature

Cross-compilation in Rust means building a binary for a different platform than the one you are currently using. Rust simplifies this with its built-in toolchains and Cargo, allowing developers to build for various architectures and operating systems with minimal setup.

To define platforms, rust uses target triple. A target triple identifies the compilation target. It follows this format:

```
<ARCH>-<VENDOR>-<OS>-<ABI>
```

For example:

```
x86_64-unknown-linux-gnu x86_64 Linux GNU
```

to list all supported targets:

```
rustc --print target-list
```

Before compiling for a different target, install the Rust standard library for that target. For example
Install support for ARM Linux (Raspberry Pi):

```
rustup target add armv7-unknown-linux-gnueabi
```

Once the target is installed, build your Rust project for that target:

```
cargo build --target=armv7-unknown-linux-gnueabi
```

Rust can compile pure Rust code easily, but if a project depends on C libraries, a cross-compiler is needed. Install cross-toolchain before compiling. In this case, we create arm binaries using Linux OS.

```
sudo apt install gcc-arm-linux-gnueabi
```

There is fastest way to resolve this issue. Instead of manually setting up compilers, use cross, which provides preconfigured Docker images for many targets.

```
cargo install cross
```

Then you can build your binaries accordingly:

```
cross build --target=<target-triple>
cross build --target=armv7-unknown-linux-gnueabi
```

Now transport your binaries for specified systems.

7) Readme.md

Unfortunately there is not a lot in readme apart from information about testing and rust version. You need to just use source code and cargo.toml files in Dockerfile.

Local

This repo assumes you have rust 1.73 installed. You may use docker, or install rust [here](#).

Build

From `../..`, run:

```
docker compose build shipping
```

Test

```
cargo test
```

8) Shipping Dockerfile

The Dockerfile uses rust:1.76 as the builder, enables multi-arch builds with `--platform=${BUILDPLATFORM}`, and defines `TARGETARCH`, `TARGETPLATFORM`, and `BUILDPLATFORM` arguments (e.g., `TARGETARCH=amd64`, `TARGETPLATFORM=linux/amd64`, `BUILDPLATFORM=linux/amd64` in my case on a Linux laptop).

```
FROM --platform=${BUILDPLATFORM} rust:1.76 AS builder
ARG TARGETARCH
ARG TARGETPLATFORM
ARG BUILDPLATFORM
RUN echo Building on ${BUILDPLATFORM} for ${TARGETPLATFORM}
```

Dockerfile runs cross-compilation check. It installs standard Rust dependencies, or for ARM64 (linux/arm64), installs `g++-aarch64` and adds Rust `aarch64` toolchain; for AMD64 (linux/amd64), installs `g++-x86-64` and adds Rust `x86_64` toolchain; errors on unsupported architectures.

```
RUN if [ "${TARGETPLATFORM}" = "${BUILDPLATFORM}" ] ; then \
    apt-get update && apt-get install --no-install-recommends -y g++ libc6-dev libprotobuf-dev \
    protobuf-compiler ca-certificates; \
    elif [ "${TARGETPLATFORM}" = "linux/arm64" ] ; then \
    apt-get update && apt-get install --no-install-recommends -y g++-aarch64-linux-gnu libc6- \
    dev-arm64-cross libprotobuf-dev protobuf-compiler ca-certificates && \
    rustup target add aarch64-unknown-linux-gnu && \
    rustup toolchain install stable-aarch64-unknown-linux-gnu; \
    elif [ "${TARGETPLATFORM}" = "linux/amd64" ] ; then \
    apt-get update && apt-get install --no-install-recommends -y g++-x86-64-linux-gnu libc6- \
    amd64-cross libprotobuf-dev protobuf-compiler ca-certificates && \
    rustup target add x86_64-unknown-linux-gnu && \
    rustup toolchain install stable-x86_64-unknown-linux-gnu; \
    else \
    echo "${TARGETPLATFORM} is not supported"; \
    exit 1; \
fi
```

Set the working directory to /app, copy the application source code (/src/shipping/ to /app/ for Rust code) and Protocol Buffers (/pb/ to /app/proto/ for gRPC).

```
WORKDIR /app/  
COPY /src/shipping/ /app/  
COPY /pb/ /app/proto/
```

Compile the Rust application (shipping): for the same platform, runs cargo build -r --features="dockerproto", and for cross-compiling, sets the correct toolchain (e.g., aarch64-linux-gnu-gcc for ARM64), build with the appropriate target, and copy the final binary to /app/target/release/shipping.

```
RUN if [ "${TARGETPLATFORM}" = "${BUILDPLATFORM}" ] ; then \  
    cargo build -r --features="dockerproto"; \  
elif [ "${TARGETPLATFORM}" = "linux/arm64" ] ; then \  
    env CARGO_TARGET_AARCH64_UNKNOWN_LINUX_GNU_LINKER=aarch64-linux-gnu-gcc \  
        CC_aarch64_unknown_linux_gnu=aarch64-linux-gnu-gcc \  
        CXX_aarch64_unknown_linux_gnu=aarch64-linux-gnu-g++ \  
    cargo build -r --features="dockerproto" --target aarch64-unknown-linux-gnu && \  
    cp /app/target/aarch64-unknown-linux-gnu/release/shipping /app/target/release/shipping; \  
elif [ "${TARGETPLATFORM}" = "linux/amd64" ] ; then \  
    env CARGO_TARGET_X86_64_UNKNOWN_LINUX_GNU_LINKER=x86_64-linux-gnu-gcc \  
        CC_x86_64_unknown_linux_gnu=x86_64-linux-gnu-gcc \  
        CXX_x86_64_unknown_linux_gnu=x86_64-linux-gnu-g++ \  
    cargo build -r --features="dockerproto" --target x86_64-unknown-linux-gnu && \  
    cp /app/target/x86_64-unknown-linux-gnu/release/shipping /app/target/release/shipping; \  
else \  
    echo "${TARGETPLATFORM} is not supported"; \  
    exit 1; \  
fi
```

Download the grpc_health_probe binary (for gRPC health checks) and make it executable

```
ENV GRPC_HEALTH_PROBE_VERSION=v0.4.24
RUN wget -qO/bin/grpc_health_probe https://github.com/grpc-ecosystem/grpc-health-probe/releases/download/${GRPC_HEALTH_PROBE_VERSION}/grpc_health_probe-linux-${TARGETARCH} && \
  chmod +x /bin/grpc_health_probe
```

Use a lightweight Debian image for production. Copy compiled Rust binary and gRPC health probe. Expose the shipping service port and run the compiled Rust binary (/app/shipping).

```
FROM debian:bookworm-slim AS release
WORKDIR /app
COPY --from=builder /app/target/release/shipping /app/shipping
COPY --from=builder /bin/grpc_health_probe /bin/grpc_health_probe
EXPOSE ${SHIPPING_PORT}
ENTRYPOINT ["/app/shipping"]
```

Kubernetes EKS: Go containers

1) Otel implementation, Checkout service explanation

This service is responsible to process a checkout order from the user. The checkout service will call many other services in order to process an order.

The OpenTelemetry SDK is initialized from main using the `initTracerProvider` function.

```
func initTracerProvider() *sdktrace.TracerProvider {
    ctx := context.Background()

    exporter, err := otlptracegrpc.New(ctx)
    if err != nil {
        log.Fatal(err)
    }
    tp := sdktrace.NewTracerProvider(
        sdktrace.WithBatcher(exporter),
        sdktrace.WithResource(initResource()),
    )
    otel.SetTracerProvider(tp)

    otel.SetTextMapPropagator(propagation.NewCompositeTextMapPropagator(propagation.TraceContext{}, propagation.Baggage{}))
    return tp
}
```

You should call `TracerProvider.Shutdown()` when your service is shutdown to ensure all spans are exported. This service makes that call as part of a deferred function in main

```
tp := initTracerProvider()
defer func() {
    if err := tp.Shutdown(context.Background()); err != nil {
        log.Printf("Error shutting down tracer provider: %v", err)
    }
}()
```

To add distributed propagation, make use of wrappers on server and client side. For example, we will show it using gRPC server and client

Server-side: Wrap your server with OpenTelemetry's gRPC interceptor.

```
grpcServer := grpc.NewServer(grpc.UnaryInterceptor(otelgrpc.UnaryServerInterceptor()))
```

Client-side: Similarly, you wrap the client connection with an interceptor to send trace data for each call.

```
conn, err := grpc.Dial("localhost:50051", grpc.WithInsecure(),  
grpc.WithUnaryInterceptor(otelgrpc.UnaryClientInterceptor()))
```

2) Spans

Within the execution of auto-instrumented code you can get current span from context.

```
span := trace.SpanFromContext(ctx)
```

Adding attributes to a span is accomplished using `SetAttributes` on the span object. In the `PlaceOrder` function several attributes are added to the span.

```
span.SetAttributes(  
    attribute.String("app.order.id", orderID.String()), shippingTrackingAttribute,  
    attribute.Float64("app.shipping.amount", shippingCostFloat),  
    attribute.Float64("app.order.amount", totalPriceFloat),  
    attribute.Int("app.order.items.count", len(prepareOrderItems)),  
)
```

Adding span events is accomplished using `AddEvent` on the span object. In the `PlaceOrder` function several span events are added. Some events have additional attributes, others do not.

Adding a span event without attributes:

```
span.AddEvent("prepared")
```

Adding a span event with additional attributes:

```
span.AddEvent("charged",
    trace.WithAttributes(attribute.String("app.payment.transaction.id", txID)))
```

3) Metrics

The OpenTelemetry SDK is initialized from main using the `initMeterProvider` function.

```
func initMeterProvider() *sdkmetric.MeterProvider {
    ctx := context.Background()

    exporter, err := otelmetricgrpc.New(ctx)
    if err != nil {
        log.Fatalf("new otel metric grpc exporter failed: %v", err)
    }

    mp :=
    sdkmetric.NewMeterProvider(sdkmetric.WithReader(sdkmetric.NewPeriodicReader
    (exporter)))
    global.SetMeterProvider(mp)
    return mp
}
```


You should call `MeterProvider.Shutdown()` when your service is shutdown to ensure all records are exported. This service makes that call as part of a deferred function in main

```
mp := initMeterProvider()
defer func() {
    if err := mp.Shutdown(context.Background()); err != nil {
        log.Printf("Error shutting down meter provider: %v", err)
    }
}()
```

In addition to manually instrumenting your application for specific metrics, OpenTelemetry also provides auto-instrumentation for the Go runtime. This allows you to collect metrics related to Go's memory usage, garbage collection, and other runtime statistics automatically.

```
err := runtime.Start(runtime.WithMinimumReadMemStatsInterval(time.Second))
if err != nil {
    log.Fatal(err)
}
```

4) Building Go apps

Install go and init your first module. It will create directory with go.mod file

```
go mod init <module-name>
```

```
myapp/
```

```
|—— go.mod  # Created by `go mod init <module-name>`
```

go.mod contains metadata about the Go module, including the module's name and dependencies

```
module <module-name>

go <go-version>

require (
    <dependency1> <version1>
    <dependency2> <version2>
    ...
)
```

To Download all dependencies and organize your project, run:

```
go mod tidy
```

Directory structure will contain go.sum now.

```
myproject/
| — go.mod (cleaned up, missing dependencies added)
| — go.sum (newly created if it didn't exist)
```

go.sum is a checksum file that ensures the integrity and security of your Go module's dependencies. It is automatically generated and maintained by Go when you run commands like

Now add source code and build application. Build determines which files to compile based on the current directory, Go module structure, and package conventions.

```
go build
```

You can run go files inside directory.

```
go run <file>.go
```

Go will read the go.mod file for dependencies, fetch any missing dependencies, execute the main.go file.

5) Readme

Readme provides information how to build go app and update dependencies:

Local Build

To build the service binary, run:

```
go build -o /go/bin/checkout/
```



Bump dependencies

To bump all dependencies run:

```
go get -u -t ./...  
go mod tidy
```



6) Checkout Dockerfile

Start a build stage using the Go 1.22 Alpine image, a minimal base with the Go environment pre-installed, optimized for building Go applications

```
FROM golang:1.22-alpine AS builder
```

Set the working directory and optimize dependency management by caching Go modules and binding local go.sum and go.mod files, enabling go mod download to fetch dependencies.

```
RUN --mount=type=cache,target=/go/pkg/mod/ \  
    --mount=type=bind,source=./src/checkout/go.sum,target=go.sum \  
    --mount=type=bind,source=./src/checkout/go.mod,target=go.mod \  
    go mod download
```

Speed up future builds by caching Go modules and build data, while --mount=type=bind mounts the local source code into the container. The go build command compiles the application, stripping debug info with -ldflags "-s -w" to reduce binary size, and outputs the binary to /go/bin/checkout/.

```
RUN --mount=type=cache,target=/go/pkg/mod/ \  
    --mount=type=cache,target=/root/.cache/go-build \  
    --mount=type=bind,rw,source=./src/checkout,target=. \  
    go build -ldflags "-s -w" -o /go/bin/checkout/ ./
```

Start with an Alpine base image, set the working directory to `/usr/src/app/`, and copy the compiled checkout binary from the builder stage into it. Expose a port defined by the `CHECKOUT_PORT` environment variable and set the checkout binary as the default executable when the container runs.

```
FROM alpine
```

```
WORKDIR /usr/src/app/
```

```
COPY --from=builder /go/bin/checkout/ ./
```

```
EXPOSE ${CHECKOUT_PORT}
```

```
ENTRYPOINT [ "./checkout" ]
```

Kubernetes EKS: JavaScript frameworks containers

1) Otel implementation, Payment service explanation

This service is responsible to process credit card payments for orders. It will return an error if the credit card is invalid or the payment cannot be processed.

It is recommended to require Node.js app using an initializer file that initializes the SDK and auto-instrumentation. When initializing the OpenTelemetry Node.js SDK in that module, you optionally specify which auto-instrumentation libraries to leverage, or make use of the `getNodeAutoInstrumentations()` function which includes most popular frameworks

```
const { NodeSDK } = require('@opentelemetry/sdk-node');
const { getNodeAutoInstrumentations } = require('@opentelemetry/auto-instrumentations-node');
const { OTLPTraceExporter, OTLPMetricExporter } = require('@opentelemetry/exporter-trace-otlp-grpc');
const { PeriodicExportingMetricReader } = require('@opentelemetry/sdk-metrics');
const { containerDetector, envDetector, hostDetector, osDetector, processDetector } = require('@opentelemetry/resources');
const { alibabaCloudEcsDetector, awsEc2Detector, awsEksDetector, gcpDetector } = require('@opentelemetry/resource-detector-*');

new NodeSDK({
  traceExporter: new OTLPTraceExporter(),
  instrumentations: [getNodeAutoInstrumentations({'@opentelemetry/instrumentation-fs': {
    requireParentSpan: true }}})],
  metricReader: new PeriodicExportingMetricReader({ exporter: new OTLPMetricExporter() }),
  resourceDetectors: [containerDetector, envDetector, hostDetector, osDetector, processDetector, alibabaCloudEcsDetector, awsEksDetector, awsEc2Detector, gcpDetector]
}).start();
```

2) Spans

Within the execution of auto-instrumented code you can get current span from context.

```
const span = opentelemetry.trace.getActiveSpan();
```

Adding attributes to a span is accomplished using `setAttributes` on the span object.

```
span.setAttributes({  
  'app.payment.amount': parseFloat(`${amount.units}.${amount.nanos}`),  
});
```

Add `addEvent` method on the current span to log events during the span's lifecycle.

```
span.addEvent('payment_received', {  
  'app.payment.amount': parseFloat(`${amount.units}.${amount.nanos}`),  
});
```

3) Metrics

Meters are created using the `@opentelemetry/api-metrics` package. Create a meter once, then reuse it to create instruments as needed. Meters and instruments should persist for efficient use.

```
const { metrics } = require('@opentelemetry/api-metrics');  
  
const meter = metrics.getMeter('paymentservice');  
const transactionsCounter = meter.createCounter('app.payment.transactions');
```

4) Build JavaScript project with npm

Initialize project

```
# Create your project directory  
mkdir my-js-project  
cd my-js-project
```

```
# Initialize npm project  
npm init -y
```

This will generate a package.json file that tracks your dependencies and scripts. If you want to add third party dependencies like Jest for unit testing you can do it using npm

```
npm install --save-dev jest
```

Or add to package.json

```
{  
  "name": "my-js-project",  
  "version": "1.0.0",  
  "description": "",  
  "main": "main.js",  
  "scripts": {  
    "test": "jest"  
  },  
  "author": "",  
  "license": "ISC",  
  "dependencies": {  
    "lodash": "^4.17.21"  
  },  
  "devDependencies": {  
    "jest": "^29.0.3"  
  }  
}
```


and install all defined dependencies using

```
npm install
```

You can install dependencies from package-lock.json file, not package.json

```
npm ci
```

If you want to run tests

```
npm test
```

5) Readme

There is not that much in Readme. Only info about demo.proto file and to use npm ci when installing dependencies

Local Build

Copy the `demo.proto` file to this directory and run `npm ci`

6) Payment Dockerfile

Set the base image for the build stage to node:22-alpine and define the working directory inside the container as `/usr/src/app/`

```
FROM node:22-alpine AS build
```

Copy the package.json and package-lock.json files from the local src/payment directory to the container, and install build tools (python3, make, g++) before running npm ci to install production dependencies.

```
COPY ./src/payment/package*.json ./
```

```
RUN apk add --no-cache python3 make g++ && npm ci --omit=dev
```

Set the base image to node:22-alpine, switch to the non-root node user, set the working directory to /usr/src/app/ and define NODE_ENV as production.

```
FROM node:22-alpine
```

```
USER node
```

```
WORKDIR /usr/src/app/
```

```
ENV NODE_ENV=production
```

Copy node_modules, application code (src/payment/), and demo.proto into the container with correct ownership for the node user..

```
COPY --chown=node:node --from=build /usr/src/app/node_modules/ ./node_modules/
```

```
COPY ./src/payment/ ./
```

```
COPY ./pb/demo.proto ./
```

Define exposed port by PAYMENT_PORT and run the application.

```
EXPOSE ${PAYMENT_PORT}
```

```
ENTRYPOINT [ "npm", "run", "start" ]
```

Kubernetes EKS: Python containers

1) Otel implementation, Recommendation service

This service is responsible to get a list of recommended products for the user based on existing product IDs the user is browsing.

This Python based service, makes use of the OpenTelemetry auto-instrumentor for Python, accomplished by leveraging the opentelemetry-instrument Python wrapper to run the scripts. This can be done in the ENTRYPOINT command for the service's Dockerfile.

```
ENTRYPOINT [ "opentelemetry-instrument", "python", "recommendation_server.py" ]
```

The OpenTelemetry SDK is initialized in the `__main__` code block. This code will create a tracer provider, and establish a Span Processor to use. Export endpoints, resource attributes, and service name are automatically set by the OpenTelemetry auto instrumentor based on environment variables.

```
tracer = trace.get_tracer_provider().get_tracer("recommendationservice")
```

2) Spans

Within the execution of auto-instrumented code you can get current span from context.

```
span = trace.get_current_span()
```

Adding attributes to a span is accomplished using `set_attribute` on the span object.

```
span.set_attribute("app.products_recommended.count", len(prod_list))
```

You can add events using the `add_event` method.

```
span.add_event("event_name", {"key1": "value1", "key2": "value2"})
```

New spans can be created and placed into active context using `start_as_current_span` from an OpenTelemetry Tracer object. When used in conjunction with a `with` block, the span will automatically be ended when the block ends execution.

```
with tracer.start_as_current_span("get_product_list") as span:
```

3) Metrics:

The OpenTelemetry SDK is initialized in the `__main__` code block. This code will create a meter provider. Export endpoints, resource attributes, and service name are automatically set by the OpenTelemetry auto instrumentor based on environment variables.

```
meter = metrics.get_meter_provider().get_meter("recommendationservice")
```

4) Building python apps with pip

Setup your working directory:

```
mkdir my-python-app  
cd my-python-app
```

It is best practice to isolate your project's dependencies using a virtual environment. This ensures that your project has its own set of libraries and does not interfere with other projects.

```
python3 -m venv venv  
source venv/bin/activate
```

Use pip (Python's package manager) to install dependencies. For example flask and requests

```
pip install flask requests
```

You can also create a requirements.txt file to track all the dependencies:

```
pip freeze > requirements.txt
```

To install dependencies listed in requirements.txt, use:

```
pip install -r requirements.txt
```

run script from terminal

```
python app.py
```

If you want to run tests, install pytest

```
pip install pytest
```

Write and run tests by simply:

```
pytest
```

For simple script execution, add the interpreter path using `#!` at the top of the script.

```
#!/usr/bin/env python3
# app.py

def add(a, b):
    return a + b

if __name__ == "__main__":
    print(add(2, 3))
```

add execution permissions to script:

```
chmod +x app.py
```

and run

```
./app.py
```

5) Readme.md

Developer told us to create protos first

Local Build

To build the protos, run from the root directory:

```
make docker-generate-protobuf
```



— • — .. •

6) Recommendation Dockerfile

Use the lightweight python:3.12-slim-bookworm image as the base image.

```
FROM python:3.12-slim-bookworm AS base
```

Create a new stage named builder based on base, install the g++ compiler without unnecessary dependencies, and clean up package lists to reduce image size.

```
FROM base AS builder
RUN apt-get -qq update \
    && apt-get install -y --no-install-recommends g++ \
    && rm -rf /var/lib/apt/lists/*
```

Set the working directory to /usr/src/app/ and copy the requirements.txt file from the local ./src/recommendation/ directory into the container's working directory.

```
WORKDIR /usr/src/app/
COPY ./src/recommendation/requirements.txt ./
```

Upgrade pip and install dependencies from requirements.txt into /reqs for isolated package management.

```
RUN pip install --upgrade pip
RUN pip install --prefix="/reqs" -r requirements.txt
```

Create a runtime stage, set the working directory, copy installed dependencies from the builder stage, and copy the application source code.

```
FROM base AS runtime
WORKDIR /usr/src/app/
COPY --from=builder /reqs /usr/local
COPY ./src/recommendation/ ./
```

Install OpenTelemetry auto-instrumentation, expose the application's port, and set the entrypoint to run the recommendation_server.py script with OpenTelemetry instrumentation.

```
RUN opentelemetry-bootstrap -a install

EXPOSE ${RECOMMENDATION_PORT}

ENTRYPOINT [ "opentelemetry-instrument", "python", "recommendation_server.py" ]
```


common troubleshooting

1) Container Image Fails to Start

Cause: Incorrect base image, missing dependencies, or application misconfiguration.

Solution: Check container logs using `kubectl logs <pod-name>`. Ensure the correct base image is used in Dockerfile and all dependencies are installed.

2) OpenTelemetry Collector Not Exporting Data

Cause: Incorrect configuration in OpenTelemetry Collector or missing exporters.

Solution: Verify the OpenTelemetry config using `kubectl describe pod <collector-pod>`. Ensure the correct exporters (e.g., OTLP, Jaeger, Prometheus) are configured.

3) Microservice Fails to Communicate with OpenTelemetry Collector

Cause: Service discovery or networking issues.

Solution: Check if the Collector is reachable using `kubectl get svc`. Ensure the correct service name and port are configured in the app's OpenTelemetry SDK settings.

4) High Latency in Tracing Data

Cause: Overloaded OpenTelemetry Collector or network latency.

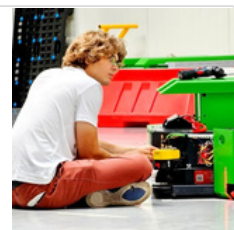
Solution: Scale the OpenTelemetry Collector with `kubectl scale deployment <collector-deployment> --replicas=<num>`. Optimize batch processing settings in the OpenTelemetry configuration.

5) Check my Kubernetes Troubleshooting series:

Michael Robotics

Hi, I'm Michal. I'm a Robotics Engineer and DevOps enthusiast. My mission is to create skill-learning platform that combats skill information overload by adhering to the set of principles: simplify, prioritize, and execute.

<https://github.com/MichaelRobotics>




Learn more about Kubernetes

Check Kubernetes and piyushsachdeva - great docs!

Setup a Multi Node Kubernetes Cluster

kubeadm is a tool to bootstrap the Kubernetes cluster

 <https://github.com/piyushsachdeva/CKA-2024/tree/main/Resources/Day27>



Kubernetes Documentation

This section lists the different ways to set up and run Kubernetes

 <https://kubernetes.io/docs/setup/>



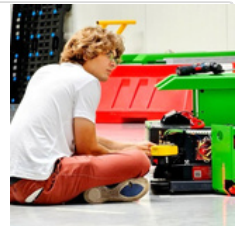
Share, comment, DM and check GitHub for scripts & playbooks created to automate process.

Check my GitHub

Michael Robotics

Hi, I'm Michal. I'm a Robotics Engineer and DevOps enthusiast. My mission is to create skill-learning platform that combats skill information overload by adhering to the set of principles: simplify, prioritize, and execute.

<https://github.com/MichaelRobotics>



PS.

If you need a playbook or bash script to manage KVM on a specific Linux distribution, feel free to ask me in the comments or send a direct message!