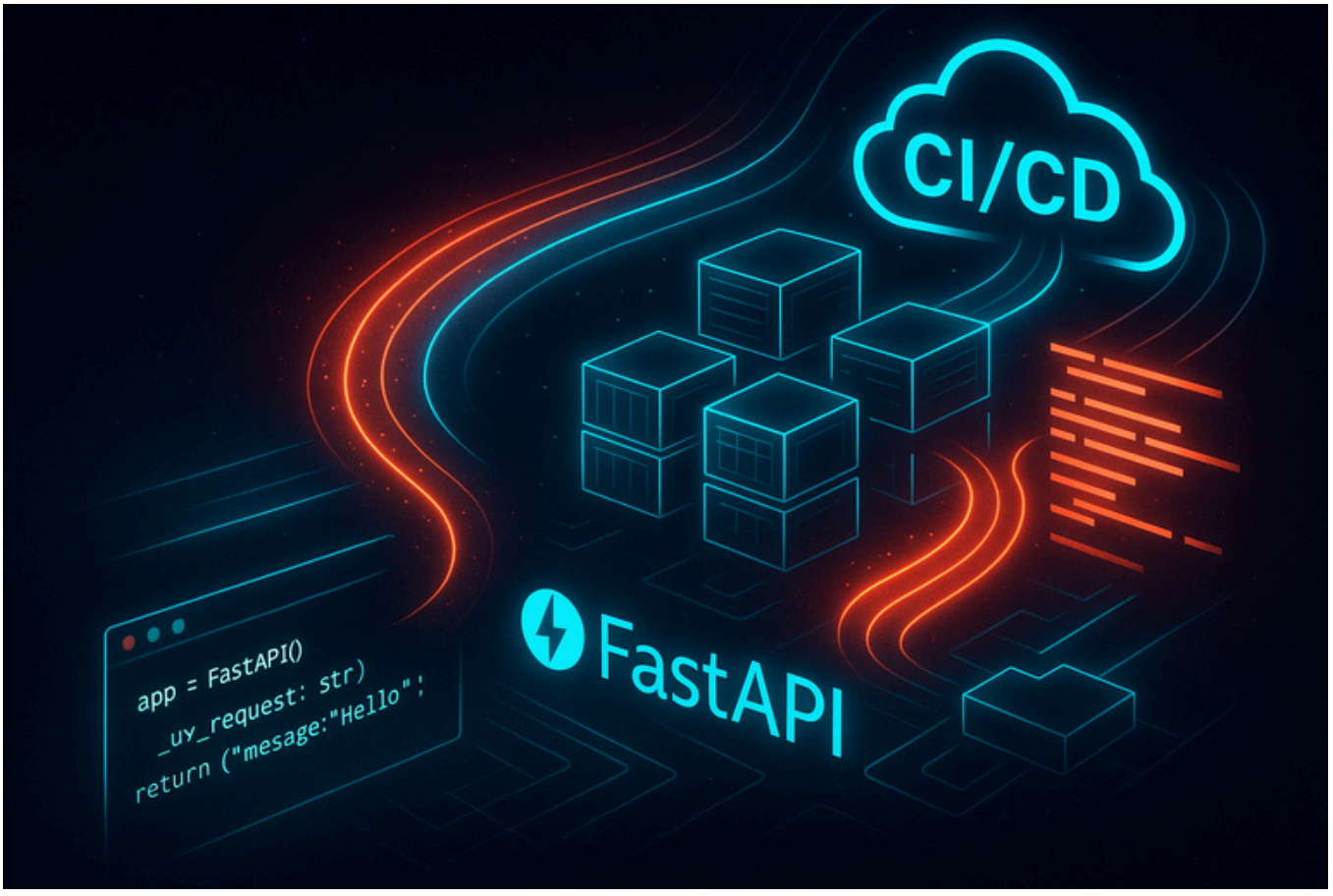
Simple CI/CD for FastAPI with Google Cloud Build and Cloud Run

davidmuraya.com/blog/fastapi-cloud-build-run-deploy-on-gcp

David Muraya January 25, 2025

Simple CI/CD for Your FastAPI App with Cloud Build and Cloud Run

6 min read



Tired of manually building and deploying your FastAPI app every time you make a change? There's a much smoother way using Google Cloud Build. Let's set up a simple Continuous Integration and Continuous Deployment (CI/CD) pipeline.

When you push code changes to your GitHub repository, this pipeline will automatically build a Docker container image and deploy it to Google Cloud Run. We'll use Cloud Build to handle the building and pushing, and Cloud Run to run our container without worrying about servers.

This means faster updates, fewer mistakes, and more time for you to focus on writing code.

What You'll Need

1. A Google Cloud Platform (GCP) project.
2. Your FastAPI application code hosted in a GitHub repository.
3. A Dockerfile in your repository to build your app's container image.
4. A cloudbuild.yaml file in your repository to tell Cloud Build what to do.

Step 1: Enable the Cloud Build API

First things first, you need to make sure Cloud Build is allowed to run in your GCP project.

1. Go to the Google Cloud Console.
2. Navigate to "APIs & Services" > "Library".
3. Search for "Cloud Build API".
4. Click on it and make sure it's "Enabled". If not, click the "Enable" button.

Step 2: Get Your Code Ready (Dockerfile and cloudbuild.yaml)

You need two key files in the root of your GitHub repository: Dockerfile and cloudbuild.yaml.

Your Dockerfile

This file tells Docker how to package your FastAPI application. Here's a sample one. Important: Replace "my-project" in the WORKDIR, COPY, and RUN commands below with the actual name of your application's directory or desired work directory name within the container.

Dockerfile

```
# Use a specific Python version. Slim-buster is a good lightweight choice.
FROM python:3.11-slim-buster

# Set the working directory inside the container
WORKDIR /my-project

# Install system packages needed (like ffmpeg if your app uses it, gcc for some Python packages)
# Update package lists, install, then clean up to keep the image small
RUN apt-get update && \
    apt-get install -y \
    gcc \
    ffmpeg \
    && rm -rf /var/lib/apt/lists/*

# Copy just the requirements file first. Docker caches this layer.
# If requirements.txt doesn't change, Docker reuses the cached layer, speeding up builds.
COPY ./requirements.txt /my-project/requirements.txt

# Install Python dependencies
RUN pip install --no-cache-dir --upgrade -r /my-project/requirements.txt

# Copy the rest of your application code
# Assumes your FastAPI code is in a directory named 'app'
COPY ./app /my-project/app

# Command to run your application when the container starts
# Uses gunicorn as the web server, binding to the port provided by Cloud Run ($PORT)
# Adjust workers/threads based on your app's needs and Cloud Run instance size
CMD exec gunicorn --bind :$PORT --workers 1 --worker-class uvicorn.workers.UvicornWorker -t 60 app.main:app
```

Your cloudbuild.yaml File

This file tells Cloud Build the steps to take: build the image, push it to Google Container Registry (GCR), and deploy it to Cloud Run.

Important: Replace "my-project" in the args for building, pushing, and deploying with the exact same name you want for your Cloud Run service and container image path.

```
steps:
  # Step 1: Build the Docker image
  # Uses the standard Docker builder provided by Google Cloud Build
  # Tags the image with the unique commit SHA for easy tracking
  - name: 'gcr.io/cloud-builders/docker'
    args: ['build', '-t', 'gcr.io/$PROJECT_ID/my-project:$COMMIT_SHA', '.']

  # Step 2: Push the built image to Google Container Registry (GCR)
  # GCR is Google's private Docker image storage
  - name: 'gcr.io/cloud-builders/docker'
    args: ['push', 'gcr.io/$PROJECT_ID/my-project:$COMMIT_SHA']

  # Step 3: Deploy the image to Cloud Run
  # Uses the gcloud command-line tool (also provided as a builder)
  # Deploys a service named "my-project" using the image we just pushed
  # Sets an environment variable (e.g., ENV=LIVE)
  # Specifies the region and project
  - name: 'gcr.io/cloud-builders/gcloud'
    args:
      - 'run'
      - 'deploy'
      - 'my-project' # This should be your service name
      - '--image=gcr.io/$PROJECT_ID/my-project:$COMMIT_SHA' # Use the image we built/pushed
      - '--set-env-vars=ENV=LIVE' # Example environment variable
      - '--region=us-central1' # Choose the region for your service
      - '--project=$PROJECT_ID'
      # Add other flags as needed: --memory, --cpu, --allow-unauthenticated, etc.

  # Step 4: Make sure 100% of traffic goes to the new version
  # This ensures the update is live immediately
  - name: 'gcr.io/cloud-builders/gcloud'
    args:
      - 'run'
      - 'services'
      - 'update-traffic'
      - 'my-project' # Your service name again
      - '--to-latest'
      - '--region=us-central1' # Same region as deployment
      - '--project=$PROJECT_ID'

# Configure Cloud Build options
options:
  # Send build logs straight to Cloud Logging
  logging: CLOUD_LOGGING_ONLY
```

Make sure both these files (Dockerfile and cloudbuild.yaml) are committed and pushed to the root of your GitHub repository.

Step 3: Set Up the Cloud Build Trigger

Now, let's connect Cloud Build to your GitHub repository.

1. In the Google Cloud Console, go to "Cloud Build" > "Triggers".
2. Click "Create trigger".
3. Give your trigger a Name (e.g., deploy-fastapi-main).
4. Under Event, choose "Push to a branch".
5. Under Source, click "Connect repository". Select "GitHub" as the source.
   You might need to authenticate with GitHub and authorize Google Cloud Build. - Select your GitHub repository from the list.

6. In the Configuration section:
    - For Branch, enter the name of the branch you want to trigger deployments from (e.g., main or master).
    - For Build configuration, select "Autodetected". Cloud Build will automatically look for your cloudbuild.yaml or Dockerfile.
7. Service Account (Important Note): By default, Cloud Build uses a standard service account. This account needs permissions to push to Container Registry and deploy to Cloud Run. If your build fails with permission errors, you might need to:
    - Create a dedicated service account with roles like "Cloud Run Admin" (roles/run.admin) and "Storage Admin" (roles/storage.admin - for GCR).
    - Go back to your trigger settings, find the "Edit Trigger" or "Service Account" section, and select your custom service account.
8. Click "Create".

Step 4: Push Your Code

That's it for setup!

Now, whenever you push a commit to the specific branch (main in our example) of your connected GitHub repository:

1. GitHub notifies Google Cloud Build.
2. Your Cloud Build trigger fires up.
3. Cloud Build reads your cloudbuild.yaml file.
4. It runs the steps: builds the Docker image, pushes it to GCR, and deploys the new version to Cloud Run, updating the traffic. You can watch the build progress in the "History" section of Cloud Build in the Google Cloud Console. Once it's green, your latest changes are live on Cloud Run!

Update - June 5, 2025: Optimizing with Multi-Stage Docker Builds

A significant improvement to both your Docker image size and Cloud Run cold start times can be achieved by implementing a multi-stage Docker build. Even when starting with a `slim` base image, a multi-stage build can reduce the final image size by more than 30%. This is because it separates the build environment (with all its dependencies like compilers and build tools) from the final runtime environment, which only contains the necessary components to run your application.

Benefits:

- **Smaller Image Size:** By discarding build-time dependencies, the final image is much leaner. This means faster uploads to your container registry and quicker downloads when Cloud Run instances scale up.
- **Improved Cold Start Times:** Smaller images generally lead to faster cold starts on serverless platforms like Cloud Run, as there's less data to pull and initialize.
- **Better Security:** Fewer packages in your final image mean a smaller attack surface.

Here's an example of an updated `Dockerfile` using a multi-stage approach:

```
# Stage 1: Builder
# This stage installs build dependencies and Python packages
FROM python:3.13-slim AS builder

WORKDIR /opt/app_build

# Install OS build dependencies required for some Python packages (e.g., gcc for C extensions, libffi-dev)
RUN apt-get update && \
    apt-get install -y --no-install-recommends \
    gcc \
    libffi-dev \
    && rm -rf /var/lib/apt/lists/*

# Copy requirements file
COPY ./requirements.txt .

# Create a virtual environment and install Python dependencies into it
# Using a venv helps keep dependencies isolated and makes the copy to the next stage cleaner
RUN python -m venv /opt/venv
ENV PATH="/opt/venv/bin:$PATH"
RUN pip install --no-cache-dir --upgrade -r requirements.txt

# Stage 2: Final image
# This stage builds the final, lean image with only runtime necessities
FROM python:3.13-slim

WORKDIR /my-app

# Install minimal runtime OS dependencies if absolutely necessary
# For a basic FastAPI app, you might not need many extra OS packages here.
# The example below includes some common ones for PDF/image generation, but adjust as needed.
# RUN apt-get update && \
#     apt-get install -y --no-install-recommends \
#     libglib2.0-0 \
#     # Add other essential runtime libs only if your app strictly requires them
#     && rm -rf /var/lib/apt/lists/*

# Copy the virtual environment (with installed packages) from the builder stage
COPY --from=builder /opt/venv /opt/venv

# Add the virtual environment's bin directory to the PATH for the runtime stage
ENV PATH="/opt/venv/bin:$PATH"

# Copy your application code
COPY ./app /my-app/app

# Set environment variable for Python (good practice for running in containers)
ENV PYTHONUNBUFFERED=1

# Command to run your application when the container starts
# Uses gunicorn as the web server, binding to the port provided by Cloud Run ($PORT)
CMD exec gunicorn --bind :$PORT --workers 1 --worker-class uvicorn.workers.UvicornWorker  --threads 4
app.main:app
```

The `ENV PYTHONUNBUFFERED=1` line is a best practice for running Python in Docker. It ensures that your application's logs are sent directly to Cloud Logging without being buffered, which prevents log loss if the container crashes.

By adopting a multi-stage build, you ensure your production container is as small and efficient as possible, directly contributing to better performance and potentially lower costs on Cloud Run.

Update - August 10, 2025: Adding Database Migrations to Your Cloud Build Pipeline

If you want to automate database migrations as part of your deployment process, you can add a migration step to your `cloudbuild.yaml`. This ensures your database schema stays in sync with your application code every time you deploy.

I've written a detailed guide on how to do this using Alembic and Google Cloud Build. Read more here: Running Database Migrations with Alembic in Google Cloud Build

This approach helps you avoid manual migration steps and keeps your deployments smooth and reliable.

Next Steps: Performance Tuning Your FastAPI Application on Cloud Run

Once your CI/CD pipeline is running smoothly, the next logical step is to optimize your application's performance on Cloud Run. This involves more than just a small Docker image; it includes tuning concurrency, managing CPU, and minimizing cold starts.

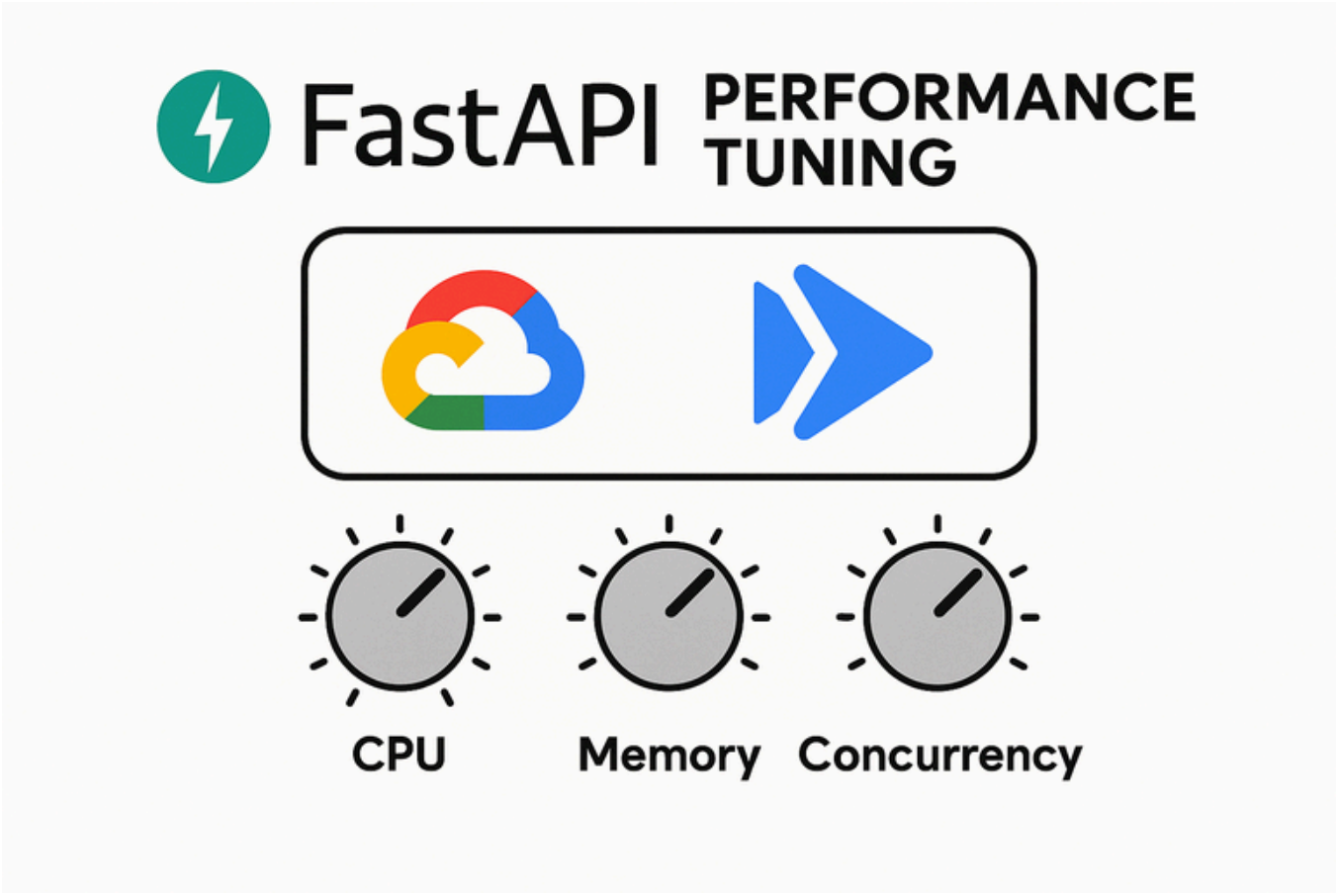
For a deep dive into these topics, read my guide: Advanced Performance Tuning for FastAPI on Google Cloud Run.

This will help you make your application faster and more cost-effective.

#python#fastapi#ci/cd#google-cloud#devops#cloud-build#cloud-run#github

David Muraya is a Solutions Architect specializing in Python, FastAPI, and Cloud Infrastructure. He is passionate about building scalable, production-ready applications and sharing his knowledge with the developer community. You can connect with him on LinkedIn.

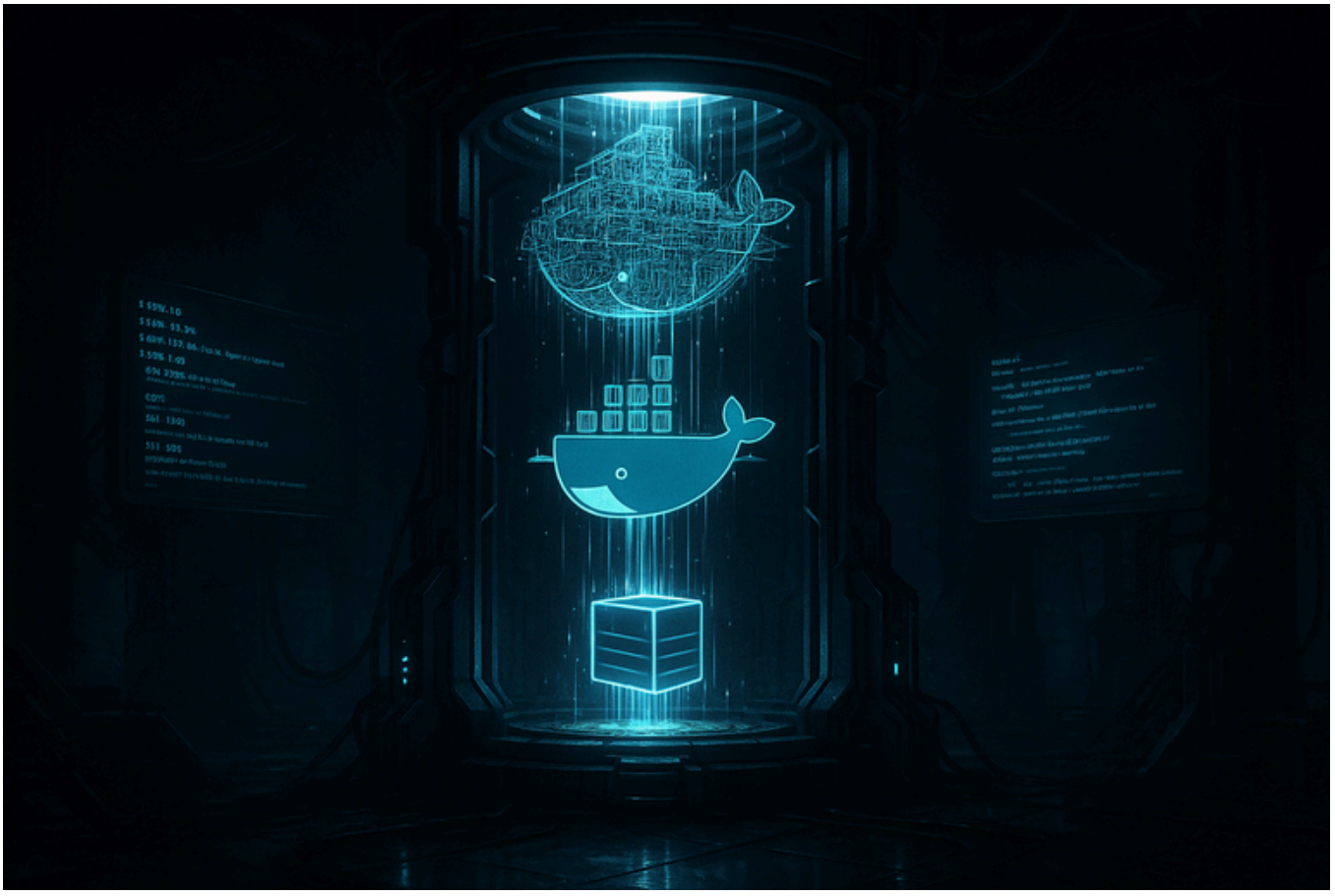
Enjoyed this blog post? Check out these related posts!



[Advanced Performance Tuning for FastAPI on Google Cloud Run]

From Cold Starts to Concurrency: A Deep Dive into FastAPI Performance on Cloud Run.

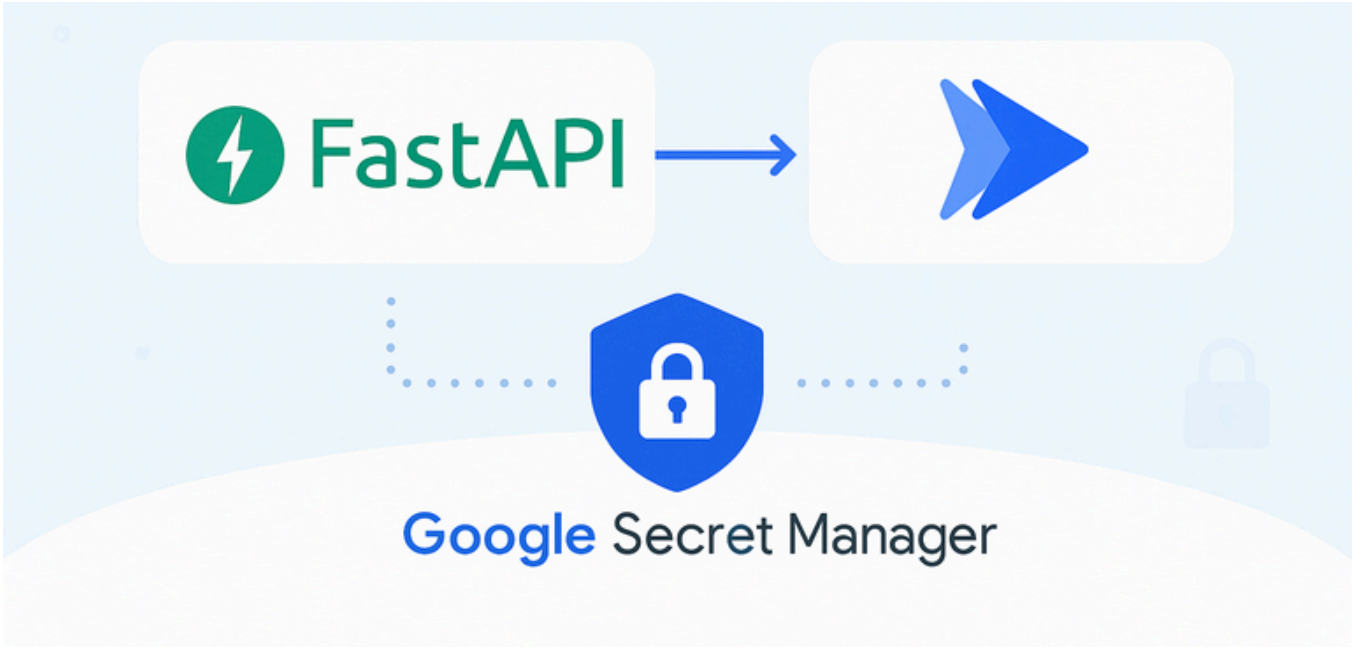
[Read More...]



[Slimmer FastAPI Docker Images with Multi-Stage Builds]

Understanding Multi-Stage Builds in Docker

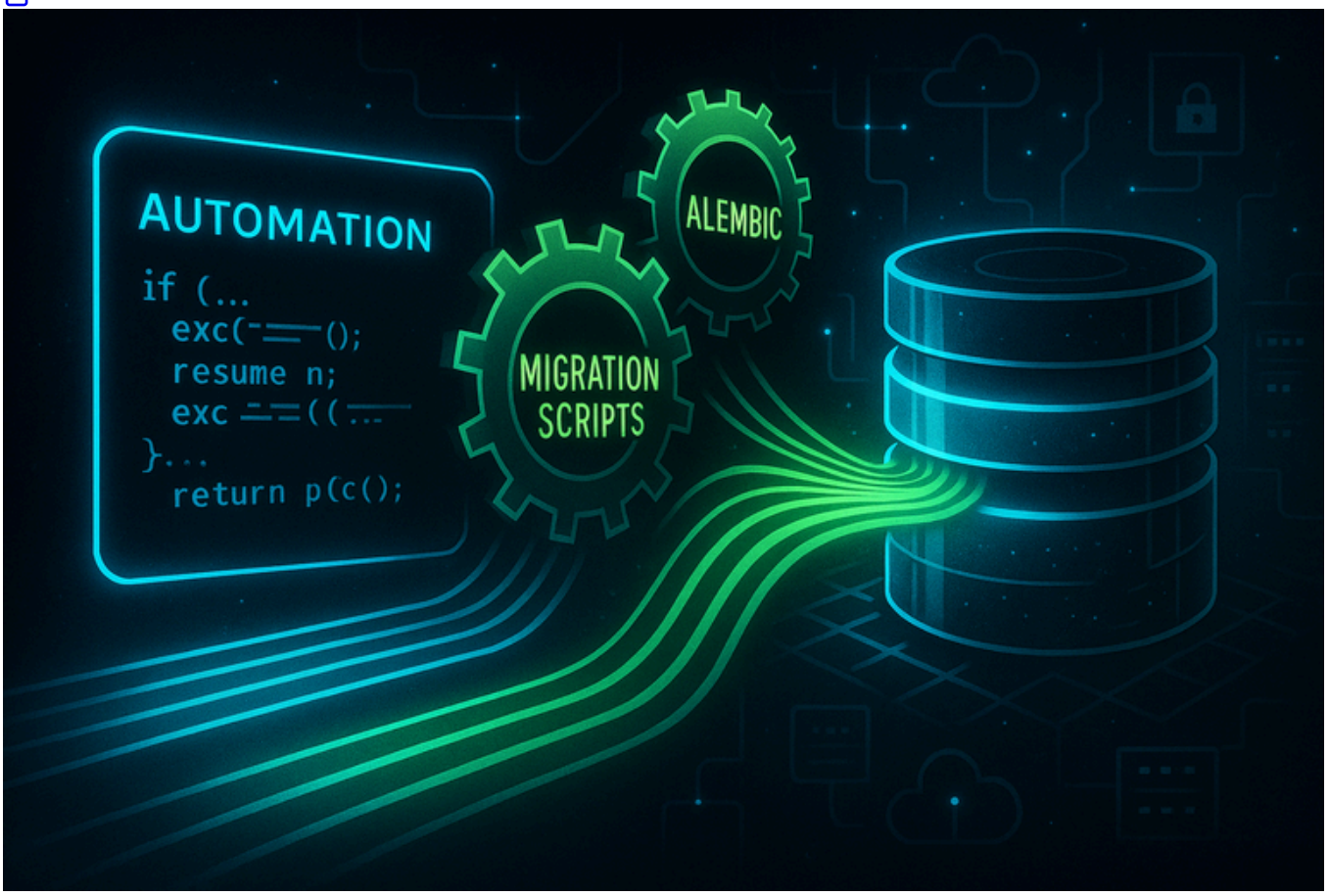
Read More...



[Secure FastAPI Environment Variables on Cloud Run with Secret Manager]

A Step-by-Step Guide to Managing Production Secrets on Google Cloud.

Read More...



Running Database Migrations with Alembic in Google Cloud Build

How to Organize and Load FastAPI Settings from a .env File Using Pydantic v2

Read More...

What You'll NeedStep 1: Enable the Cloud Build APIStep 2: Get Your Code Ready (Dockerfile and cloudbuild.yaml)Your DockerfileYour cloudbuild.yaml FileStep 3: Set Up the Cloud Build TriggerStep 4: Push Your CodeUpdate - June 5, 2025: Optimizing with Multi-Stage Docker BuildsUpdate - August 10, 2025: Adding Database Migrations to Your Cloud Build PipelineNext Steps: Performance Tuning Your FastAPI Application on Cloud Run

Back to Blogs

Have a project in mind? Send me an email at hello@davidmuraya.com and let's bring your ideas to life. I am always available for exciting discussions.