Kaunas University of Technology

Faculty of Informatics

Distributed Systems and Algorithms

Lab report 2

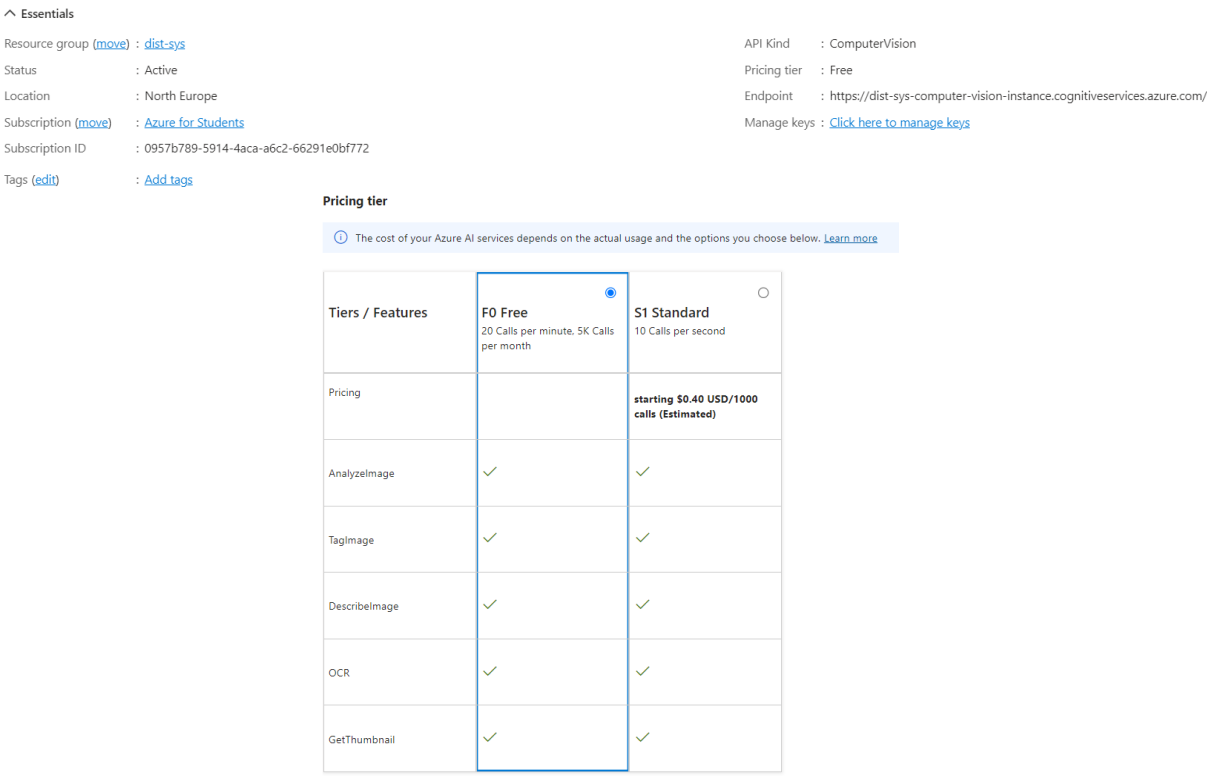
Ruben Vandamme
Project author

Kaunas, 2024

1. Business logic

My variant number is 0. The app I build allows the user to upload an image. It will be analyzed by Azure AI vision to get a summarizing sentence for that image. Next, an image will be generated for this sentence using the stable diffusion version 1.5 model running locally. Thus, the user will be able to generate similar images using an already existing image.

2. WebAPI + connection to Azure AI Vision



Essentials

Resource group (move)	: dist-sys	API Kind	: ComputerVision
Status	: Active	Pricing tier	: Free
Location	: North Europe	Endpoint	: https://dist-sys-computer-vision-instance.cognitiveservices.azure.com/
Subscription (move)	: Azure for Students	Manage keys	: Click here to manage keys
Subscription ID	: 0957b789-5914-4aca-a6c2-66291e0bf772		
Tags (edit)	: Add tags		

Pricing tier

The cost of your Azure AI services depends on the actual usage and the options you choose below. Learn more

Tiers / Features	F0 Free 20 Calls per minute, 5K Calls per month	S1 Standard 10 Calls per second
Pricing		**starting $0.40 USD/1000 calls (Estimated)**
AnalyzeImage	✓	✓
TagImage	✓	✓
DescribeImage	✓	✓
OCR	✓	✓
GetThumbnail	✓	✓

In the above screenshots, the Azure AI Vision setup details are shown.

Most things are already said in the video (see zip file). In the remainder of this report, I will focus on things that were not yet mentioned, as to avoid wasting the time of both the reader and the writer.

```
if __name__ == "__main__":
    global stub
    global templates

    # gRPC dbservice
    channel = grpc.insecure_channel('localhost:5002')
    stub = dbservice_pb2_grpc.DBServiceStub(channel)

    # app setup
    templates = Jinja2Templates(directory="templates")
    uvicorn.run(app, host="127.0.0.1", port=5000)
```

Here you can see the stub setup for connection to the gRPC server. I use Jinja2 to provide the index.html file and Bootstrap for a standardized webpage visualization. FastAPI is used as the framework and Uvicorn for running the server.

```python
@asynccontextmanager
async def lifespan(app: FastAPI):
    global unfinished_tasks

    unfinished_tasks = []

    task = asyncio.create_task(periodic_task())
    try:
        yield
    finally:
        task.cancel()
        try:
            await task
        except asyncio.CancelledError:
            pass

        for task_id in unfinished_tasks:
            print(f"Finalizing task {task_id} during shutdown.")
            try:
                stub.Finalize(dbservice_pb2.FinalizeRequest(error=True))
            except grpc.RpcError as e:
                print(f"Failed to finalize task {task_id}. Database will have inconsistent state!")
```

Here it is shown how the periodic task was set up. To prevent blocking, we yield. If something were to go wrong (like exiting the server), unfinished tasks are captured and handled accordingly. If something were to go wrong with the gRPC server, who is responsible for the database, the database will have an inconsistent state. As setting up a fail-safe database is not the assignment here, I left it at that.

3. gRPC service

The gRPC service I implemented is responsible for all database actions. Separating this functionality from the web API has all the typical benefits of separation-of-concern.

```python
db = load_db()
def on_exit():
    global db
    save_db(db)

atexit.register(on_exit)
```

Upon starting the service, the database gets loaded. Upon exiting the service, it gets saved.

```python
def serve():
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    dbservice_pb2_grpc.add_DBServiceServicer_to_server(DBServiceServicer(), server)
    server.add_insecure_port('[::]:5002')
    server.start()
    print("gRPC server is running on port 5002...")
    try:
        server.wait_for_termination()
    except KeyboardInterrupt:
        print("Server interrupted.")

if __name__ == '__main__':
    serve()
```

Here you can see how the files generated from the .proto file are being used to set up the service and serve it to port 5002.

```python
class DBServiceServicer(dbservice_pb2_grpc.DBServiceServicer):
    def Add(self, request, context):
        id = add(db, request.img_bytes)
        return dbservice_pb2.AddResponse(id=id)
```

Here is an example of how the service provides a function to be called remotely.

4. XML web service

```
model = StableDiffusionPipeline.from_pretrained(
    "runwayml/stable-diffusion-v1-5",
    cache_dir="./model_cache"
)
model.to("cpu")
```

Above, the model for generating images is provided.

```
image = model(
    prompt,
    #num_inference_steps=25,
    #height=256,
    #width=256
).images[0]
```

Here the model gets called. We could reduce the number of iterations the model goes through or decrease the image size. The best results, however, are with the default parameters and those take a long time.

```
@app.post("/generate_image", response_class=Response)
async def generate_image(request: str = Body(..., media_type="application/xml")):
```

Finally, the signature of the endpoint is shown. It accepts only xml media types.