Distributed Scalable Face Recognition System

Dokumentacija

Kolegij: Distribuirani sustavi
Fakultet informatike u Puli

Autor: Antonio Labinjan
Mentor: doc. dr. sc. Nikola Tanković

December 16, 2025

Contents

1. Sažetak

Ova dokumentacija opisuje dizajn i implementaciju distribuiranog, skalabilnog sustava za prepoznavanje lica u realnom vremenu temeljenog na modernim metodama računalnog vida i umjetne inteligencije. Sustav se sastoji od više udaljenih nodeova (edge kamera) koji lokalno izvode ekstrakciju značajki lica koristeći CLIP model, te centralnog servera koji koristi FAISS za vektorsko pretraživanje i klasifikaciju osoba. Arhitektura omogućuje horizontalno skaliranje, load balancing i failover mehanizme između nodeova, čime se postiže visoka pouzdanost i rad u stvarnom vremenu.

Komunikacija između nodeova i servera odvija se putem Redis posrednika, koji služi kao message broker i load balancing sloj. Redis omogućuje asinkronu razmjenu embeddinga, redoslijedno procesiranje zahtjeva i stabilan prijenos podataka čak i u slučajevima privremene mrežne nestabilnosti.

Na strani nodea implementiran je mehanizam shouldClassify() koji optimizira prijenos podataka - embedding se šalje prema serveru samo ako je detekcija dovoljno različita ili ako je prošlo određeno vrijeme od zadnjeg slanja. Time se značajno smanjuje mrežno opterećenje bez gubitka performansi sustava.

Sustav također uključuje unknown alert system koji prepoznaje višestruke pokušaje neuspjele identifikacije s različitih nodeova u kratkom vremenskom periodu, uz potpuno poštivanje GDPR smjernica i bez otkrivanja identiteta korisnika.

Rješenje pokazuje kako se principi distribuiranih sustava mogu učinkovito primijeniti na prepoznavanje lica u stvarnom vremenu, uz naglasak na skalabilnost, efikasnost i privatnost u edge-to-server arhitekturi.

2. Opis aplikacije

Razvijeni sustav predstavlja distribuirano rješenje za prepoznavanje lica u stvarnom vremenu koje kombinira snagu compter visiona i distribuiranih sustava. Sastoji se od više **nodeova** (edge kamera) koji lokalno obrađuju video u stvarnom vremenu, te centralnog **servera** koji objedinjuje rezultate, vrši klasifikaciju i vodi evidenciju detekcija. Također, vrši se dvostruka segmentacija i uklanjanje pozadine i na server-side djelu i na nodevima kako bi se smanjio utjecaj pozadine na klasifikaciju. U trenutnoj verziji, server je implementiran u FastAPI-ju, dok su nodevi implementirani u čistom Pythonu te koriste ugrađene kamere računala i dodatne vanjske kamere (trenutno USB priključak). Nodeovi koriste **CLIP** model za ekstrakciju značajki lica (embeddinga), dok server koristi **FAISS** za vektorsko pretraživanje i usporedbu embeddinga s postojećom bazom poznatih osoba. Komunikacija između nodeova i servera odvija se putem **Redis** posrednika koji služi kao message broker i load balancing sloj, čime se omogućuje skalabilna i asinkrona razmjena podataka.

Sustav podržava horizontalno skaliranje - dodavanjem novih nodeova automatski se povećava kapacitet obrade bez promjene konfiguracije servera. Osim toga, uključeni su **failover mehanizmi** koji omogućuju nastavak rada i u slučaju kvara pojedinog nodea ili pada servera. Nodesi su lako zamjenjivi i lako ih je dodati (potrebno je samo spojiti kameru i pokrenuti Python script). Ukoliko server padne, embeddinzi se svejedno šalju u Redis te mogu biti dohvaćeni kad server opet proradi.

Jedna od prednosti sustava je mehanizam `should_classify()`, koji značajno smanjuje promet u mreži jer osigurava da se embedding šalje samo kad je stvarno potreban (promjena lica ili protek vremena). Time se postiže optimalna ravnoteža između performansi i učinkovitosti.

Ciljano tržište i korisnici

Primarna ciljna skupina su **organizacije, poduzeća i institucije** kojima je potreban automatizirani, brzi i pouzdani sustav za identifikaciju i evidenciju osoba. Sustav se može koristiti u različitim scenarijima:

- **Evidencija prisutnosti zaposlenika** u tvrtkama, laboratorijima ili fakultetima.
- **Sigurnosni nadzor** u poslovnim i javnim prostorima, s mogućnošću detekcije nepoznatih osoba (unknown alert system).
- **Autonomni ulazni sustavi** koji reagiraju na prepoznato lice i odobravaju pristup bez potrebe za fizičkim kontaktom.

Za razliku od centraliziranih rješenja, ovaj sustav pruža **distribuiranost, otpornost i privatnost** – obrada se odvija lokalno na nodeovima, a prema serveru se šalju samo embeddingi, ne i slike lica. To znači da je sustav u skladu s **GDPR** smjernicama i može se primijeniti u okruženjima s visokim sigurnosnim i etičkim zahtjevima. Time se postiže moderna kombinacija umjetne inteligencije i distribuiranih principa koja otvara put prema **lokalnim edge-to-server** rješenjima spremnima za buduće proširenje u cloud okruženje.

Analiza tržišta i konkurencija

Motivacija za razvoj ovog projekta proizašla je iz prethodnog rada na jednostavnim web aplikacijama baziranim na arhitekturi **1-client–1-server**. Takav pristup bio je dovoljan za manje projekte, ali pokazao je svoja ograničenja u situacijama gdje je potrebno obraditi veći broj ulaza

(kamera) i postići real-time sinkronizaciju između više uređaja. Cilj je bio unaprijediti postojeću aplikaciju pretvaranjem je u **distribuirani sustav**, sposoban za obradu podataka s više kamera u stvarnom vremenu uz istovremeno održavanje stabilnosti, točnosti i brzine.

Glavni motiv je bio **automatizirati brojne svakodnevne procese bilježenja prisutnosti** — eliminirati ručni unos, liste i kartice te omogućiti sustavu da sam prepoznaje osobu pri ulasku ili izlasku. Takav pristup ne samo da štedi vrijeme, već i uklanja ljudske pogreške te omogućuje analitiku u stvarnom vremenu.

Na tržištu već postoje sustavi za prepoznavanje lica, poput **Azure Face API-ja**, **AWS Rekognitiona** ili **OpenCV/DeepFace** rješenja, ali svi oni imaju svoja ograničenja:

- **Cloud ovisnost:** većina rješenja zahtijeva konstantnu internetsku vezu i vanjsku infrastrukturu, što stvara ovisnost o trećim servisima.
- **Privatnost:** podaci o licima šalju se na vanjske servere, što je neprihvatljivo u GDPR osjetljivim okruženjima.
- **Centraliziranost:** tradicionalna rješenja koriste jedan server, što otežava skaliranje i povećava rizik od kvara.

Razvijeni sustav uklanja te probleme uvođenjem **distribuirane arhitekture** s lokalnom obradom podataka na **edge nodeovima** i centralnim serverom koji upravlja klasifikacijom i sinkronizacijom putem **Redis** brokera. Time je postignut balans između brzine, sigurnosti i fleksibilnosti, bez ovisnosti o vanjskim servisima.

Table 1: SWOT analiza distribuiranog sustava za prepoznavanje lica

Snage (S)	**Slabosti (W)**
• Distribuiran i skalabilan sustav • Lokalna obrada podataka (GDPR-friendly) • Real-time prepoznavanje i logging • Modularnost i lako dodavanje novih nodeova	• Zahtjevnija implementacija i konfiguracija • Potrebno održavanje više uređaja
Prilike (O)	**Prijetnje (T)**
• Primjena u pametnim zgradama i industriji 4.0 • Integracija s postojećim sustavima kontrole pristupa • Potencijal za komercijalizaciju i SaaS model	• Brz razvoj konkurentskih AI rješenja • Moguće regulatorne promjene (GDPR, AI Act) • Tehnički problemi s pouzdanošću mreže

Projekt time ne samo da rješava konkretan problem evidencije prisutnosti, već otvara vrata prema **budućim edge-to-server sustavima** koji kombiniraju računalni vid, distribuirano računalstvo i automatizaciju poslovnih procesa. Ovo rješenje demonstrira kako se napredni AI modeli mogu učinkovito raspodijeliti između rubnih uređaja i centralnog sustava, što predstavlja važan korak prema modernim, skalabilnim i etičkim sustavima prepoznavanja lica.

Distribuirani sustav za prepoznavanje lica sastoji se od centralnog servera, Redis queuea i više udaljenih čvorova (nodeova), pri čemu svaki čvor predstavlja kameru ili modul za snimanje lica. Čvorovi ekstraktiraju embeddinge lica pomoću CLIP modela i odlučuju, kroz funkciju `should_classify()`, kada će poslati embedding serveru, čime se optimizira mrežni promet. Svi embeddingi se šalju preko Redis queuea na centralni server, koji upravlja FAISS indeksom i provodi klasifikaciju te algoritam glasanja kako bi odredio identitet osobe. Server zatim vraća rezultate natrag čvorovima po potrebi, a cjelokupan sustav omogućava dinamičko dodavanje

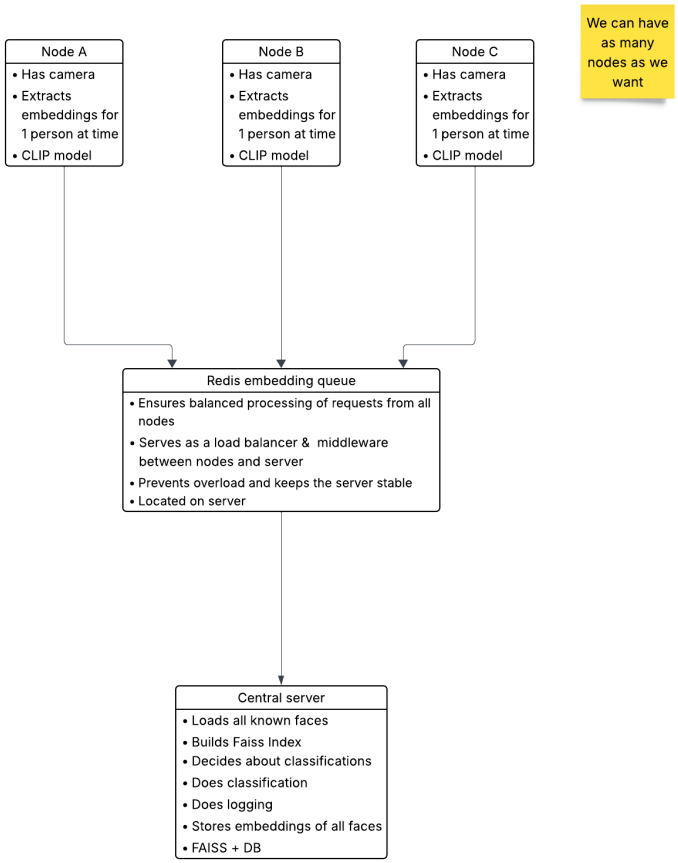


Figure 1: Shema arhitekture sustava

novih nodeova i podešavanje parametara u realnom vremenu, dok shema vizualno prikazuje tok podataka i veze između nodeova, Redis-a i servera.

1 4. Razrada funkcionalnosti *(max. 8–15 stranica)*

1. Funkcionalnosti servera

- Pokretanje FastAPI servera s REST API endpointima
- Učitavanje dataset-a lica iz foldera
- Dodavanje pojedinačnih lica u dataset i ekstrakcija CLIP embeddinga
- Normalizacija i spremanje embeddinga u memoriju
- Gradnja FAISS indeksa za brzu pretragu najbližih susjeda
- Klasifikacija lica pomoću FAISS indeksa i CLIP embeddinga
- Testiranje različitih threshold vrijednosti za klasifikaciju
- Praćenje neprepoznatih pokušaja (Unknown) i intruder alert sustav
- Logiranje detekcija u memoriju
- Globalno logiranje preko Redis streama
- Redis queue za slanje embeddinga u worker thread
- Worker thread koji čita embeddinge iz Redis queue-a i klasificira ih
- Dead-letter queue u Redisu za neispravne ili višestruko neuspjele poruke
- Pametno odlučivanje kada klasificirati embedding po node-u (`should_classify`)
- Praćenje posljednjeg embeddinga i timestamp-a po node-u
- API endpointi za pregled loga detekcija u JSON i HTML formatu
- API endpointi za pregled queue-a, threshold statistike i aktivnih node-ova
- Vizualizacija threshold statistike i aktivnih node-ova u HTML tablicama
- API endpointi za intruder alert log u JSON i HTML formatu
- Endpoint za ponovno učitavanje dataset-a i rebuild FAISS indeksa
- Ping endpoint za provjeru zdravlja servera
- Logging svih važnih operacija s vremenskim oznakama

Funkcionalnosti Node-a

- Učitavanje tokena iz JSON datoteke
- Povezivanje s Redis serverom i provjera konekcije
- Postavljanje osnovnih parametara Node-a (NODE_ID, threshold distance i vrijeme)
- Praćenje posljednjeg embeddinga i vremena po node-u
- Inicijalizacija CLIP modela i procesora za ekstrakciju embeddinga lica
- Inicijalizacija Haar Cascade detektora lica

- Inicijalizacija Mediapipe Face Mesh-a za preciznu segmentaciju lica
- Segmentacija lica iz frame-a koristeći Face Mesh
- Odlučivanje treba li klasificirati novi embedding (`should_classify`)
- Praćenje frame-ova koji su previše tamni i ignoriranje u slučaju prevelike tamnosti
- Otvaranje kamere i provjera da je dostupna i radi (health check)
- Postavljanje rezolucije kamere (640x480)
- Nepravilno dohvaćeni frame-ovi se preskaču
- Detekcija lica u svakom frame-u koristeći Haar Cascade
- Segmentacija i ekstrakcija embeddinga iz svakog lica
- Normalizacija embeddinga
- Slanje embeddinga u Redis queue za daljnju obradu
- Logiranje svih važnih događaja i grešaka
- Vizualno označavanje detektiranih lica i status poruka na frame-u
- Praćenje performansi: FPS i prosječna latencija svakih 30 frame-ova
- Spremanje latencije u tekstualnu datoteku
- Omogućavanje prekida programa tipkom 'q' i čišćenje resursa pri zatvaranju

Funkcionalnosti Dijagrami (use case, class) Prototip

2 5. Implementacija *(max. 3–5 stranica)*

Za potrebe jednostavnog i pouzdanog pokretanja distribuiranog sustava za prepoznavanje lica korišten je `docker-compose`, koji omogućuje definiranje i orkestraciju svih komponenti sustava unutar jedinstvene konfiguracijske datoteke. Ovakav pristup osigurava da se FastAPI poslužitelj, Redis posrednik i svi pomoćni servisi pokreću u potpuno konzistentnim i reproducibilnim okruženjima, neovisno o softveru instaliranom na lokalnoj mašini. Svaki servis se izolira u vlastiti kontejner s točno određenim verzijama biblioteka i ovisnosti, čime se eliminiraju problemi vezani uz konfiguraciju sustava, verzijske konflikte i ručne instalacije. Pokretanje cjelokupne infrastrukture svodi se na jednu naredbu, što drastično ubrzava razvoj, testiranje i implementaciju, a istovremeno jamči da će sustav jednako raditi na svakom računalu ili poslužitelju.

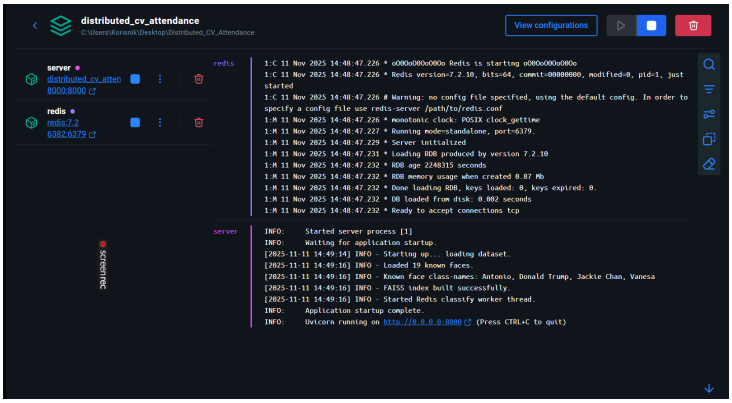


Figure 2: Docker compose okruženje pokrenutog sustava koje se sastoji od FastApi servera i Redisa

2.1 Dockerfile

```
FROM python:3.12

WORKDIR /app

RUN apt-get update && apt-get install -y \
    libgl1 \
    libglib2.0-0 \
 && rm -rf /var/lib/apt/lists/*

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

EXPOSE 8000

CMD ["uvicorn", "server_FASTAPI:app", "--host", "0.0.0.0", "--port", "8000"]
```

Dockerfile prikazan u sustavu služi za kontejnerizaciju FastAPI servisa unutar distribuiranog sustava. Kao osnovna slika koristi se `python:3.12`, čime se osigurava konzistentno i reproducibilno izvršno okruženje neovisno o krajnjem sustavu. Radni direktorij unutar kontejnera postavljen je na `/app`, što omogućuje jasno strukturiran raspored aplikacijskog koda. Radi podrške computer

visionu i grafičkim ovisnostima, instaliraju se sistemske biblioteke `libgl1` i `libglib2.0-0`, koje su nužne za ispravno izvođenje OpenCV i srodnih paketa u headless okruženju. Python ovisnosti definirane su u datoteci `requirements.txt` te se instaliraju korištenjem `pip`-a bez zadržavanja lokalnog cachea, čime se smanjuje konačna veličina Docker slike. Cjelokupni izvorni kod aplikacije zatim se kopira u kontejner, a mrežni port `8000` izlaže se kako bi servis bio dostupan ostalim komponentama distribuiranog sustava. Pokretanje aplikacije ostvaruje se putem ASGI poslužitelja `Uvicorn`, koji omogućuje asinkrono rukovanje zahtjevima i skalabilnu komunikaciju između distribuiranih čvorova.

Za razliku od serverskih komponenti, kamere odnosno *node* procesi pokreću se lokalno na fizičkim uređajima, izvan Docker okruženja. Glavni razlog za takav pristup je činjenica da pristup fizičkoj kameri unutar kontejnera često nije pouzdan niti podržan na svim operacijskim sustavima, što bi moglo otežati inicijalizaciju video streama i liveness detekciju. Osim toga, lokalno izvršavanje omogućuje da se obrada slike i ekstrakcija značajki obave s minimalnom latencijom, bez dodatnih mrežnih slojeva između kamere i procesorske logike. Time se postiže veća responzivnost sustava i smanjuje opterećenje glavnog poslužitelja, jer se prema Redis posredniku šalju samo već obrađeni embeddingi, a ne sirovi video podaci. Ovakva arhitektura također omogućuje fleksibilno skaliranje - broj lokalnih kamera može se povećavati bez potrebe za mijenjanjem centralne infrastrukture, što sustav čini iznimno prilagodljivim i skalabilnim u realnim uvjetima implementacije.

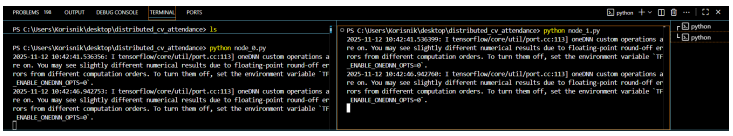
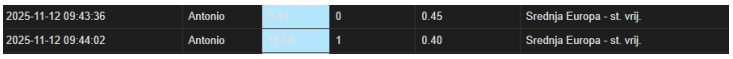


Figure 3: Pokretanje nodesa kroz terminal

Kao što je vidljivo na slici, za dodavanje novog nodea u sustav, dovoljno je fizički spojiti kameru i pokrenuti kod. Jedina manja komplikacija očituje se u tome što je svakoj novoj kameri potrebno podesiti indeks i ID u kodu. Koristi se pravilo po kojem je ID jednak indeksu kamere i nazivu node fajla što olakšava snalaženje i brzo i efikasno upravljanje kamerama.

Sam server ima izrazito jednostavnan grafički interface koji se sastoji od tablice logova u formatu: timestamp, name, score, node ID, threshold i timezone. Timestamp - predstavlja točan datum i vrijeme detekcije (s preciznošu od sekudne) Name - predstavlja ime osobe koja je detektirana Node ID - predstavlja unique identifikator svakog nodea Threshold - predstavlja optimalni threshold koji je iskorišten za detekciju Timezone - predstavlja vremensku zonu u kojoj se node vrti i time omogućava rad na širokom području (kad bi se kamere teoretski spojile bežično)



2025-11-12 09:43:36	Antonio	0.40	0	0.45	Srednja Europa - st. vrij.
2025-11-12 09:44:02	Antonio	0.462	1	0.40	Srednja Europa - st. vrij.

Figure 4: Primjer logginga

Interface na nodesima je također minimalan. Pri pokretanju nodea otvara se cv2 window koji prikazuje feed trenutne kamere te stvara bounding box oko detektiranih lica. Pošto se klasifikacija odvija na serveru te zbog privatnosti, na node feedu se ne vidi identitet osobe (ne zapisuje se rezultat klasifikacije koji je poznat samo serveru).

Za asinkronu komunikaciju između distribuiranih čvorova i centralnog poslužitelja implementiran je `Redis` koji u ovom sustavu djeluje kao middleware i load balancer. Svaki čvor, nakon što generira *embedding* lica pomoću CLIP modela, šalje podatke u zajednički `Redis` queue u kojem se privremeno pohranjuju svi zahtjevi za klasifikaciju. Centralni poslužitelj istodobno čita te zapise i obrađuje ih pomoću FAISS baze, čime se postiže dinamičko balansiranje opterećenja

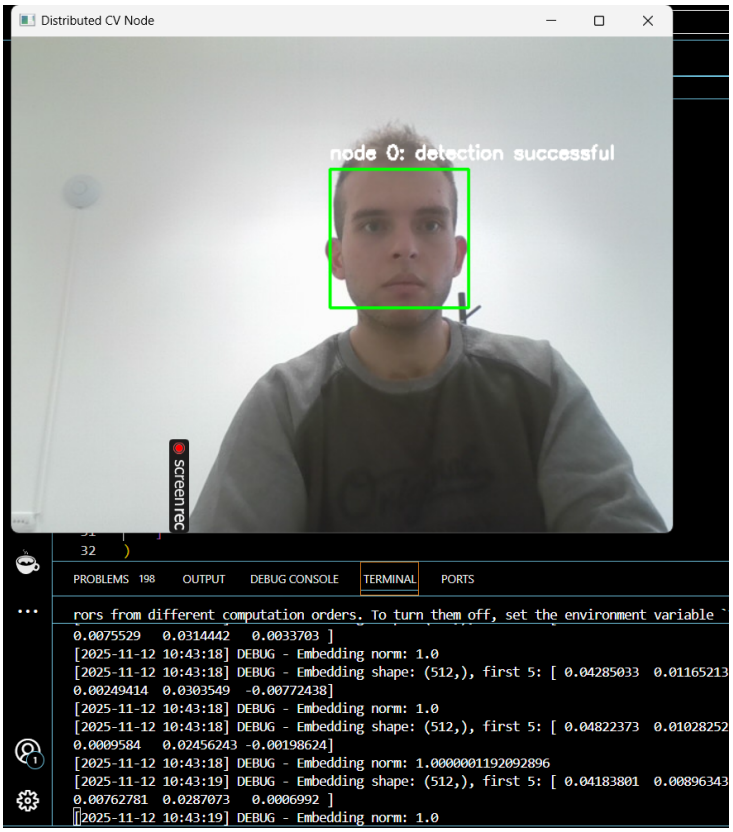


Figure 5: Primjer detekcije

između više aktivnih čvorova i poslužiteljskih procesa. Ovakva arhitektura omogućuje skalabilan i otporan sustav: broj čvorova i poslužitelja može se slobodno povećavati bez potrebe za promjenom osnovne logike komunikacije. Redis time učinkovito eliminira bottleneckove i omogućuje paralelnu, distribuiranu obradu podataka u stvarnom vremenu.

Uloga Pydantic-a U sustavimu, Pydantic model služi kao ključni mehanizam za validaciju ulaznih podataka. Na primjer, model `EmbeddingRequest` osigurava da svaki primljeni embedding bude lista realnih brojeva (`list[float]`), dok `node_id` bude cijeli broj s defaultnom vrijednošću 0. Time se sprječava slanje neispravnih ili nekompatibilnih podataka u FAISS pretragu ili mehanizam glasanja, što povećava robusnost i pouzdanost sustava. Osim toga, Pydantic automatski parsira JSON requeste u Python objekte, smanjujući potrebu za ručnom konverzijom tipova, a u kombinaciji s FastAPI-em olakšava generiranje precizne dokumentacije API-ja, omogućujući frontend aplikacijama da točno znaju što poslati serveru.

Sam hardverski setup je prilično jednostavan. Sastoji se od centralnog računala na kojem se pokreće server i kamera (nodesa):

Pošto se u trenutnoj implementaciji koriste usb kamere, ograničenje takvog sustava je svakako limitiran broj kamera.

3 Arhitektura, klase i komponente

Ovaj distribuirani sustav sastoji se od dva glavna dijela: *node-side* (radni čvorovi s kamerama) i *server-side* (centralni sustav). Svaki dio sadrži specifične module i klase koje zajedno omogućuju ekstrakciju značajki, slanje embeddinga, klasifikaciju, logiranje i nadzor sustava.

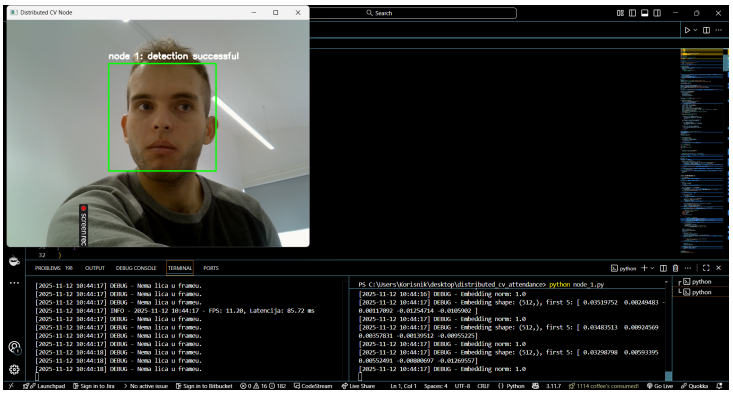


Figure 6: Primjer detekcije

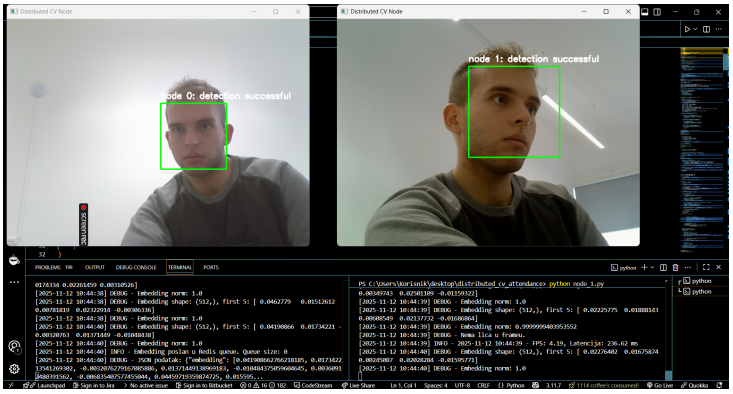


Figure 7: Primjer detekcije s više kamera

3.1 Node-side Komponente

3.1.1 Camera / Capture Modul

- **CameraReader** – odgovoran za dohvat frameova ili event-based podataka.
- **FrameGrabber** – obavlja manipulaciju frameova i rezanje lica.

3.1.2 Embedding Modul (CLIP Encoder)

- **ClipEncoder**
  - `encode_face(image)` – generira embedding lica.

3.1.3 Should-Classify Mehanizam

- **ShouldClassify**
  - uloga: sprečava nepotrebno slanje embeddinga.
  - pamti: zadnji poslani embedding po čvoru, vrijeme slanja.

3.1.4 Queue Sending Modul

- **RedisPublisher**
  - `publish_embedding(embedding, node_id)`

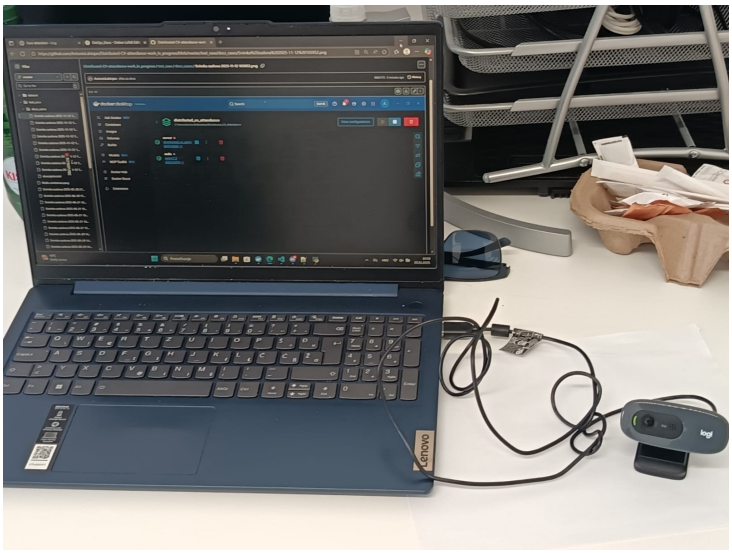


Figure 8: Primjer detekcije s više kamera

3.2 Server-side Komponente

3.2.1 Redis Queue Consumer / Worker

- **RedisConsumer**
  - sluša queue i prosljeđuje embeddinge klasifikatoru.

3.2.2 Feature Matcher i Klasifikator (FAISS + Voting)

- **FaissIndexManager**
  - upravlja FAISS indexom.
  - `search(embedding, k)` – vraća najbližih k susjeda.
- **VotingClassifier**
  - implementira k1/k2 glasanje.
  - vraća: ID osobe, confidence, distance.
- **ThresholdManager**
  - dinamičko prilagođavanje decision thresholda.

3.2.3 Logging Modul

- **DetectionLogger**
  - zapisuje sve detekcije (node, osoba, vrijeme, distance

3.2.4 Monitoring i Dashboard

- **StatsCollector** – prikuplja metrike sustava.
- **DashboardAPI** – služi za prikaz statistika, logova i stanja nodeova.

3.3 Infrastructure i Background Komponente

3.3.1 Config Modul

- **Config / Settings**
  - centralizirana konfiguracija sustava: putanje modela, Redis URL, pragovi, parametri votinga itd.

3.3.2 Healthcheck i Failover

- **NodeHealthMonitor**
  - provjerava aktivnost nodeova.
- **FailoverManager**
  - upravlja fallback scenarijima i prebacivanjem opterećenja.

3.4 Feature Store / Dataset Komponente

- **EmbeddingStore** – pohranjuje embeddinge korisnika.
- **FaceCropper** – odgovoran za detekciju i segmentaciju lica.

3.5 API Sloj

- Implementiran preko FastAPI-a.
  - `/register_user`
  - `/update_embedding`
  - `/get_logs`
  - `/stats`
  - `/threshold/update`

4 REST API Endpoints

Sustav koristi REST API implementiran pomoću *FastAPI* frameworka. U nastavku je pregled dostupnih endpointa, njihove namjene i povratnih podataka.

Figure 9: Pregled API ruta

4.1 /redis-test

Method: GET
Provjera ispravnosti Redis veze. Endpoint služi kao health-check za Redis instancu.

4.2 /log

Method: GET
Vraća sistemski log u sirovom (tekstualnom ili JSON) obliku, namijenjen za programatsku analizu i debugiranje.

4.3 /log/html

Method: GET
HTML prikaz sistemskog loga, optimiziran za pregled u pregledniku.

4.4 /

Method: GET
Početni endpoint aplikacije. Koristi se kao osnovni health-check ili landing endpoint.

4.5 /queue_contents

Method: GET
Dohvaća trenutno stanje Redis queuea, uključujući sve zaprimljene, ali još neobrađene stavke.

4.6 /threshold_stats

Method: GET
Prikazuje statistiku vezanu uz threshold mehanizme klasifikacije (npr. aktivni pragovi, povijesne promjene i frekvenciju prilagodbi).

4.7 /ping

Method: GET
Minimalni endpoint za provjeru dostupnosti servera. Vraća jednostavan odgovor bez dodatne obrade.

4.8 /active_nodes/html

Method: GET
HTML prikaz trenutno aktivnih čvorova u distribuiranom sustavu, uključujući njihove statuse i zadnje vrijeme aktivnosti.

4.9 /intruder_alerts

Method: GET
Vraća popis detektiranih intruder događaja u strukturiranom formatu pogodnom za daljnju obradu.

4.10 /intruder_alerts/html

Method: GET
HTML vizualizacija intruder alertova, namijenjena ljudskom nadzoru sustava.

4.11 /reload_dataset

Method: GET
Ručno ponovno učitavanje dataset-a za klasifikaciju bez potrebe za restartom servera.

Kako je isprogramirano Detaljno objasnit Koje su klase i komponente unutar aplikacije Dijagrami, use case, class

5 6. Korisničke upute *(max. 4–6 stranica)*

5.1 Instalacija / Pokretanje

5.2 Korištenje - korak po korak

Figure 10: Screenshot - primjer korištenja.

Stavit slike setupa (hardverskega)

Reference

References

[1] Primjer reference.

A Dodatak A: Tehničke specifikacije

B Dodatak B: Logovi i testovi