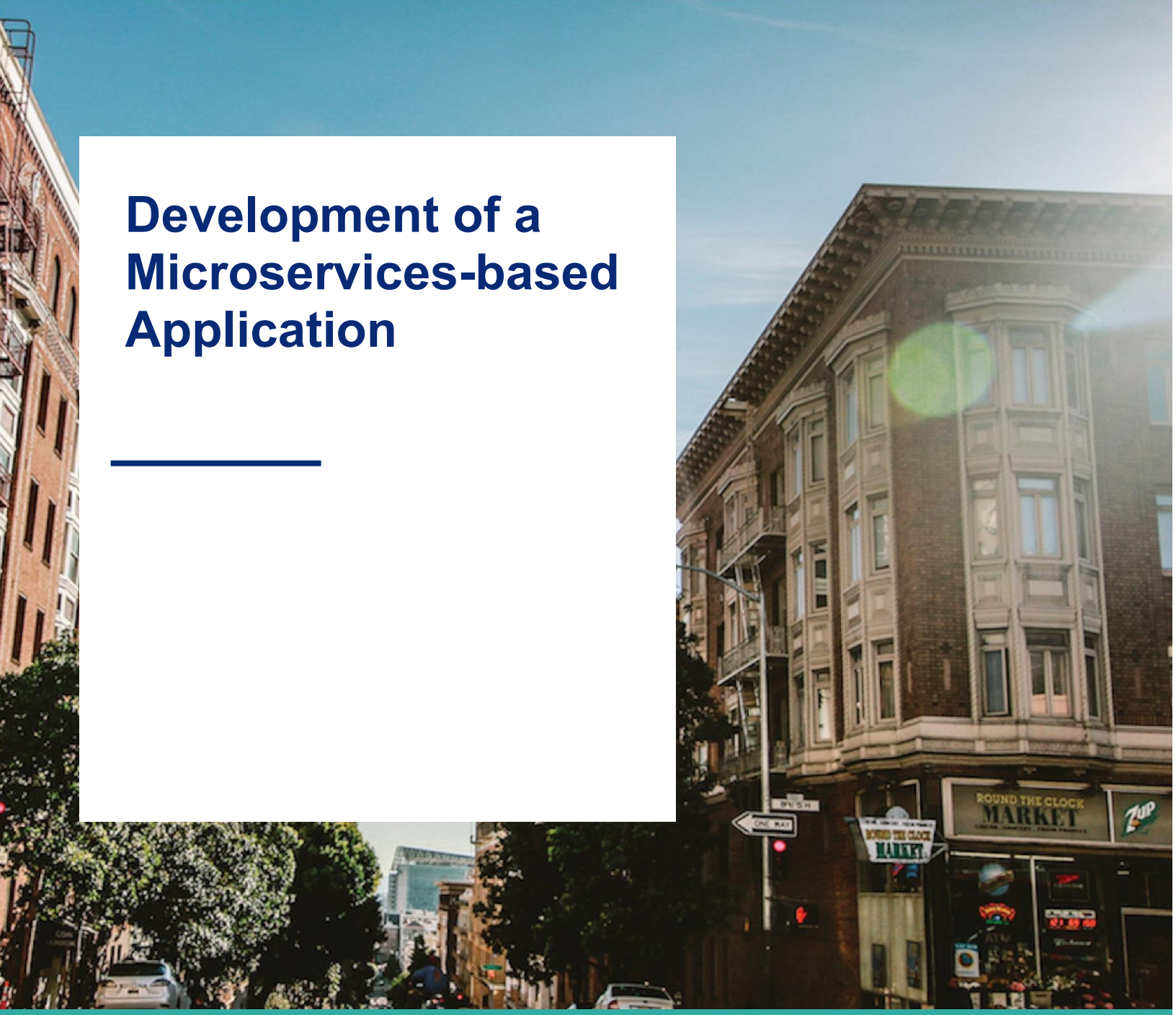
Development of a Microservices-based Application



JANUARY 6

Authored by: Michail Angelos Karvelas

1.Executive Summary

This report documents the design, implementation, and deployment of a secure microservices-based application for document redaction. The application system is designed to identify and mask Personally Identifiable Information (PII) such as names, email addresses, and street addresses from text and PDF documents. The solution proposed in the present report adopts a distributed architecture utilizing a Python Fast API backend (Producer) and a Client Interface (Consumer), orchestrated via Consul for service discovery**.**

The project covers and analyses two key phases: Part A focuses on the development of RESTful microservices and their interaction, while Part B addresses containerization, cloud clustering on Azure, and advanced DevOps implementation including rolling updates.

2. System Architecture (Part A)

2.1 MICROSERVICES PATTERN

The application follows a decoupled Producer Consumer pattern to ensure separation of concerns:

- **Producer Service (Redaction API):** A Python based REST API built with **FastAPI**. It handles the CPU intensive tasks of Natural Language Processing (NLP) and Optical Character Recognition (OCR).
- **Consumer Service (Client):** A decoupled interface that interacts with the backend. It acts as the entry point for user requests, forwarding data to the producer.
- **Service Discovery (Consul):** HashiCorp Consul is employed to manage service registration. This allows the Consumer to dynamically discover the Producer's location without relying on hardcoded IP addresses

2.2 SERVICE DISCOVERY IMPLEMENTATION

To satisfy the requirement for dynamic service location, the Redaction Service utilizes the python-consul2 library**.**

- **Registration:** On startup, the service registers its IP and Port (8000) with the Consul Agent.
- **Health Check:** It exposes a root route (/) that Consul pings to verify availability.
- **Discovery:** The Client queries Consul for the service ID redaction-service to resolve the current active address.

3. Implementation Details

I began the project by prioritizing backend development using Python, FastAPI, and NLP. For the frontend, I utilized Electron with HTML and CSS to create a local interface for quick testing. However, during the cloud deployment phase on Azure, I discovered that the desktop-based Electron architecture was incompatible with the required web-based container environment, a constraint I had not anticipated initially. In order to resolve this issue, I refactored the frontend by replacing the Electron desktop shell with an Express.js web server and converting the internal IPC calls to standard HTTP fetch requests. This allowed the application to run smoothly within a Docker container and be accessed via a web browser.

3.1 The Hybrid Redaction Logic

Pattern Matching (Regex): is a sequence of characters that forms a search and is used for patten matching

- *Emails:* Matches standard formats (e.g., user@hotmail.com).
- *Street Addresses:* Identifies patterns starting with digits followed by suffixes (e.g., "123 Main St").

Natural Language Processing (NLP): is used for entity recognition
via SpaCy (en_core_web_sm model):

- **Names**: Identifying people (PERSON).
- **Locations**: Identifying cities, countries, Islands and states (GPE, LOC).
- **Organizations**: Identifying companies and institutions (ORG).
- **Facilities**: Identifying buildings and facilities (FAC)

The backend employs a hybrid detection strategy combining Regular Expressions (Regex) and Natural Language Processing (spaCy) to maximize accuracy. While Regex is highly effective at identifying deterministic patterns—such as email addresses or phone numbers—it lacks the ability to understand context. For example, a regex script cannot easily distinguish between 'Park' as a surname and 'park' as a location. To bridge this

gap, I integrated spaCy, which excels at context-aware Named Entity Recognition (NER), allowing the system to correctly identify names and locations within sentences. By layering these two technologies, the system mitigates the weaknesses of both approaches: Regex handles the structured data that NLP might miss in isolation, while NLP catches the complex entities that Regex cannot define.

3.2 Advanced API Maturity (HATEOAS & Hashing)

To meet requirements for Level 3 REST Maturity, the following were implemented:

- **Hashing:** A POST /hash endpoint computes the SHA-256 fingerprint of the text for integrity verification.
- **HATEOAS:** Every redaction response includes dynamic navigational links (_links), allowing the client to discover related actions (like hashing) programmatically.

4. Cloud Deployment Strategy (Part B)

4.1 Containerization

Both microservices were containerized using **Docker** to ensure consistency between the development and production environments.

- **Backend:** The Python service was packaged using a python:3.12-slim image. Crucially, system-level dependencies (tesseract-ocr, poppler-utils, libgl1) were installed via apt-get to support the visual redaction features.
- **Orchestration:** A docker-compose.yml file was used for local testing to simulate the distributed environment before cloud deployment.

4.2 Cluster Infrastructure (Azure Kubernetes Service)

To meet the requirement for a scalable cluster architecture with explicit node management, the application was deployed to **Azure Kubernetes Service (AKS)**.

- **Cluster Implementation:** I provisioned an AKS cluster named redaction-cluster.
- **Node Configuration:** The cluster was configured with a **Node Pool of 2 Nodes** (Standard_B2s virtual machines). This physical separation ensures that the application runs across distinct servers, satisfying the hardware redundancy and "Cluster of 2 nodes" requirement.

4.3 Kubernetes Services & Networking

To manage network traffic and connectivity within the cluster, I implemented **Kubernetes Services**:

- **External Access (LoadBalancer):** The Frontend/Consumer service is exposed using a Service of type LoadBalancer. This prompts Azure to provision a public IP address, allowing external users to access the web interface securely over the internet.
- **Internal Communication (ClusterIP):** The Redaction Backend is exposed via a ClusterIP Service. This creates a stable internal IP address accessible only within the cluster. The Frontend discovers the Backend using this internal DNS name (e.g., http://redaction-backend), ensuring secure, private communication between microservices without exposing the API directly to the public web.

4.4 Scaling Strategy

To verify the system's ability to handle high loads, the Redaction Service was configured for horizontal scaling.

- **Replicas:** A Kubernetes Deployment manifest was defined with replicas: 3.
- **Result:** The orchestrator maintains three active copies (Pods) of the Redaction Microservice distributed across the 2 nodes. If one Pod crashes or a Node becomes unresponsive, traffic is instantly routed to the remaining healthy instances, guaranteeing High Availability.

4.5 CI/CD & Rolling Updates

To satisfy the requirement for automated, zero-downtime updates, a Continuous Deployment pipeline was implemented using **GitHub Actions**.

- **Workflow:** The pipeline is defined in .github/workflows/azure-deploy.yml.
- **Trigger:** Every push to the main branch triggers the build process.
- **The Update Mechanism:**
    1. **Build:** GitHub Actions builds the new Docker image and pushes it to the **Azure Container Registry (ACR)**.
    2. **Deploy:** The pipeline executes kubectl set image (or uses the Azure K8s Set Context action) to update the Deployment.
    3. **Rolling Update:** Kubernetes performs a native **Rolling Update**. It incrementally launches new Pods with the V2 image and terminates old V1 Pods only after the new ones pass their readinessProbe. This guarantees zero downtime for users during the update process.

5. Critical Analysis & Challenges

5.1 Challenge: Visual Redaction in Containers

Problem: The application worked locally but crashed on Docker with FileNotFoundError: tesseract.

Analysis: Python libraries like pytesseract are wrappers around binary tools. A standard Python container does not include these OS-level tools.

Solution: I modified the Dockerfile to execute apt-get update && apt-get install -y tesseract-ocr. This highlights the distinction between application dependencies (pip) and system dependencies (apt)

5.2 Challenge: NLP False Negatives

Problem: The NLP model initially failed to redact capitalized locations like "USA" or "California" consistently.

Analysis: The model classified these as ORG (Organization) or LOC (Location), while my initial code only filtered for GPE.

Solution: I expanded the entity filter to include GPE, LOC, FAC, and ORG and implemented case-insensitive matching (.lower()). This significantly improved the recall rate for address detection.

5.3 Limitation: Database State

Analysis: To this coursework, I used a local **SQLite** database (sqlite:///./redaction.db) to log usage stats. In a production environment with multiple replicas, this file is not shared, leading to inconsistent stats across instances.
Mitigation: In a real-world scenario, this would be replaced by a managed database (e.g., Azure SQL) to ensure data persistence and consistency across all cluster nodes.

6. Conclusion

The developed solution successfully demonstrates the complete microservices lifecycle. By combining robust backend processing (OCR/NLP) with a cloud-native architecture on Azure, the project satisfies the functional requirements of data security while also adhering to operational best practices (Service Discovery, Scaling, and Rolling Updates). The implementation validates the effectiveness of Azure Container Apps as a modern, Kubernetes-based platform for deploying secure microservices.