

POLITECNICO DI TORINO

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA
INFORMATICA



Tesi di Laurea Magistrale

**Riconoscimento di dispositivi di
protezione individuale in ambito
industriale tramite infrastruttura
cloud**

Relatore:
Prof. Luca Ardito

Candidato:
Rei Zoto

Dicembre 2024

Sommario

Indice

Sommario	3
Elenco delle figure	6
Elenco delle tabelle	8
1 Introduzione	10
2 Background	13
2.1 Infortuni, Sicurezza Industriale e DPI	13
2.2 Computer Vision	19
2.3 Cloud Computing nell'Industria	23
2.4 Lavori Correlati	25
3 Tecnologie e Servizi	29
3.1 Docker	29
3.2 MQTT	33
3.3 Protocolli di streaming locale	36
3.4 Amazon Web Services	41
3.5 Apache Flink	47
4 Implementazione del sistema	53
4.1 Architettura	53
4.2 Edge Runtime e Ingestion	57
4.3 Preprocessing e trasformazione dei dati	60
4.4 Big data analysis	62
4.5 Deployment	66

5 Risultati	71
5.1 Approccio	71
5.2 Test case e analisi	72
5.3 Limitazioni	74
5.4 Lavori futuri	75
Conclusioni	77
A Codice	79
A.1 Dockerfile	79
A.2 Kinesis Video Streams Consumer Lambda	82
A.3 Apache Flink Stream Process Job	87
Bibliografia	91

Elenco delle figure

2.1	Infortuni sul lavoro accertati positivi per genere e modalità di accadimento nell'anno 2022	13
2.2	infortuni in occasione di lavoro accertati positivi per settore di attività nell'anno 2022	14
2.3	onere economico complessivo stimato (approccio bottom up)	15
2.4	stima dei costi complessivi approccio top down	16
2.5	Modello del neurone artificiale sulla base del funzionamento di un neurone biologico.	19
2.6	Gatto di Hubel e Wiesel[5]. Studi che hanno permesso di dare una definizione di recettore, scoprire l'organizzazione gerarchica della corteccia visiva nei mammiferi e formalizzare il concetto di retinotopia.	21
2.7	Lenet-5(1998). Primo modello ad aver dimostrato l'efficacia delle reti convoluzionali (CNN) nella comprensione delle immagini e ha aperto la strada a molte delle architetture moderne di deep learning. Crediti: Fei-Fei Li, CS231(Stanford).	23
2.8	Prestazioni Learning Factory: custom dataset.	26
2.9	Sistema per la gestione della sicurezza in fabbrica.	27
2.10	Risultati del sistema di rilevazione.	28
3.1	Confronto tra macchine virtuali tradizionali e containers. . .	29
3.2	Esempio di condivisione di layers in Docker. Le due immagini risultanti hanno la stesso sistema di base, condividendo lo stesso ambiente (Debian) e runtime (Python).	32
3.3	Schema architettura publish-subscribe.	33

3.4	Flusso pacchetti di controllo per QoS2. I livelli più semplici funzionano in maniera simile, con un numero di pacchetti inferiore. Si può notare come il disacoppiamento fornito dal broker faciliti lo scambio dei messaggi.	35
3.5	Effetto del jitter nella riproduzione dello streaming. Senza buffering i pacchetti non vengono riprodotti seguendo la stessa temporizzazione della sorgente.	37
3.6	Stack di protocolli per lo streaming real-time.	38
3.7	Classificazione ad alto livello dei servizi AWS. Accessibili attraverso API, si dividono nelle due macro-aree hardware e software.	43
3.8	Rilevamento tramite Rekognition dei dispositivi di sicurezza individuali.	46
3.9	Esempio di funzionamento del runtime in Apache Flink.	49
3.10	Dataflow livello logico.	50
3.11	Dataflow livello fisico.	50
4.1	Schema dello use case.	54
4.2	Architettura ad alto livello del sistema.	54
4.3	Sottosistema di ingestion e preprocessing.	56
4.4	Moduli di processamento e generazione degli allarmi.	57
4.5	Pubblicazione messaggio MQTT sul servizio AWS IoT via console.	60
4.6	Diagramma modello dati.	64
4.7	Dataflow logico del sistema.	65
4.8	Stack set dell'infrastruttura di sistema.	67
4.9	Dettagli deployment sul gateway.	69
5.1	Latenza totale del sistema.	73

Elenco delle tabelle

3.1 Classificazione dei servizi AWS di interesse.	45
5.1 Test case.	72

Capitolo 1

Introduzione

La sicurezza sul lavoro rappresenta un elemento fondamentale all'interno dell'industria manifatturiera, dove l'interazione tra macchinari complessi e operai espone a numerosi rischi. Come noto, gli infortuni sul lavoro nel settore manifatturiero sono tra i più frequenti e gravi, con conseguenze significative sia per i lavoratori che per le aziende. Garantire un ambiente di lavoro sicuro non solo tutela la salute e il benessere dei dipendenti, ma contribuisce anche a migliorare la produttività e a ridurre i costi associati agli incidenti. Essi infatti possono comportare gravi conseguenze per i lavoratori, inclusi infortuni permanenti, invalidità e, in casi estremi, decessi. Tali incidenti non solo influiscono sulla qualità della vita dei dipendenti e delle loro famiglie, ma comportano anche ripercussioni economiche rilevanti per le aziende. I costi diretti includono spese mediche e indennità di infortunio, mentre i costi indiretti comprendono la perdita di produttività, la necessità di sostituzione del personale e i danni alla reputazione aziendale. Oltre alle conseguenze dirette sugli individui, gli incidenti sul lavoro hanno un impatto economico significativo sulle aziende e sulla società nel suo complesso. Le aziende devono affrontare spese legali, aumenti dei premi assicurativi e potenziali sanzioni normative in caso di inadempienza alle leggi sulla sicurezza. Inoltre, la perdita di fiducia dei consumatori e dei partner commerciali può influenzare negativamente le performance finanziarie e la competitività dell'azienda sul mercato. Sul piano sociale, gli incidenti sul lavoro contribuiscono a un aumento dei costi sanitari e riducono la produttività nazionale. La società nel suo complesso subisce un impatto economico derivante dalla perdita di forza lavoro qualificata e dall'aumento delle richieste di assistenza sociale. Pertanto, investire nella sicurezza sul lavoro rappresenta non solo

un obbligo etico e legale, ma anche una strategia economica vantaggiosa a lungo termine.

I Dispositivi di Protezione Individuale (DPI) sono strumenti essenziali per prevenire gli incidenti sul lavoro e ridurre l'esposizione dei lavoratori a rischi specifici. DPI comuni includono caschi, guanti, occhiali protettivi, maschere respiratorie e indumenti resistenti agli agenti chimici. L'uso corretto e costante dei DPI è fondamentale per garantire la sicurezza dei lavoratori, ma la loro efficacia dipende dalla conformità e dalla corretta applicazione delle normative da parte dei dipendenti. Inoltre, monitorare l'uso dei DPI in ambienti industriali può risultare complesso, soprattutto in contesti ad alta dinamicità e con elevati volumi di produzione. Tradizionalmente, questo monitoraggio è stato effettuato attraverso ispezioni manuali, che possono essere dispendiose in termini di tempo e risorse, e soggette a errori umani. Pertanto, vi è una crescente necessità di soluzioni automatizzate e tecnologicamente avanzate per garantire un controllo efficace e continuo dell'utilizzo dei DPI. L'innovazione tecnologica ha aperto nuove prospettive per migliorare la sicurezza sul lavoro nell'industria manifatturiera. In particolare, la computer vision e il cloud computing emergono come strumenti potenti per automatizzare il rilevamento dei DPI e monitorare in tempo reale le condizioni di sicurezza.

La **computer vision** permette alle macchine di interpretare e analizzare immagini e video, identificando automaticamente la presenza e l'uso corretto dei DPI. Attraverso algoritmi di deep learning, i sistemi di computer vision possono riconoscere oggetti specifici, come caschi e guanti, e verificare la loro corretta indossatura da parte dei lavoratori. Questo approccio non solo aumenta l'efficienza del monitoraggio, ma riduce anche la dipendenza da interventi manuali, minimizzando gli errori e garantendo una supervisione costante e accurata. Il **cloud computing**, d'altra parte, fornisce l'infrastruttura necessaria per gestire e analizzare grandi quantità di dati provenienti dai sistemi di computer vision. Attraverso piattaforme cloud, è possibile archiviare, elaborare e accedere ai dati in modo scalabile e flessibile, permettendo una gestione centralizzata e accessibile delle informazioni sulla sicurezza. Inoltre, il cloud computing facilita l'integrazione con altri sistemi aziendali, consentendo una visione completa delle operazioni e una risposta tempestiva agli incidenti rilevati. L'integrazione di computer vision e cloud computing rappresenta quindi una svolta nel campo della sicurezza industriale, offrendo soluzioni avanzate per il monitoraggio dei DPI e la prevenzione degli incidenti.

In questo contesto, la presente tesi si propone di sviluppare un sistema basato su Amazon Rekognition, un servizio di computer vision offerto da Amazon Web Services (AWS), per il rilevamento automatico dei DPI nell'industria manifatturiera. L'obiettivo principale è quello di generare una infrastruttura scalabile per l'analisi di dati semistrutturati e non strutturati all'interno di una fabbrica. In particolare, dato un insieme di macchinari, come ad esempio bracci robotici, si vuole ottenere il controllo dell'effettivo indossamento dei dispositivi di sicurezza da parte degli operatori (operai, manutentori) all'interno di uno stabilimento, in modo tale da garantirne loro la sicurezza sul posto di lavoro.

Capitolo 2

Background

2.1 Infortuni, Sicurezza Industriale e DPI

In questa sezione si vedranno delle statistiche relative agli infortuni sul lavoro, quale sia la risposta normativa al problema della sicurezza industriale dal punto di vista degli attori coinvolti, integrata con la definizione di dispositivi di sicurezza. Questo tema è di primaria importanza per garantire non solo la salute e il benessere dei lavoratori, ma anche l'efficienza operativa e la sostenibilità economica delle aziende. Secondo i dati forniti dall'Istituto Nazionale per l'Assicurazione contro gli Infortuni sul Lavoro (INAIL), nel 2022 il settore manifatturiero ha registrato un tasso di infortuni del 13,9% sul totale[1].

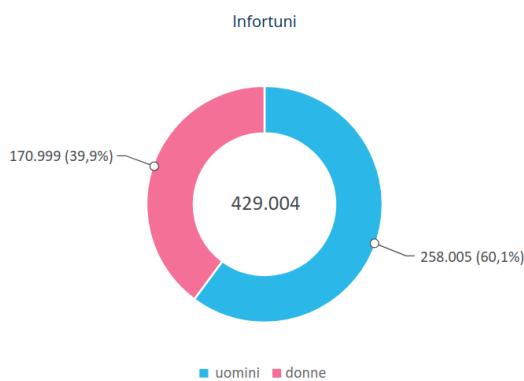
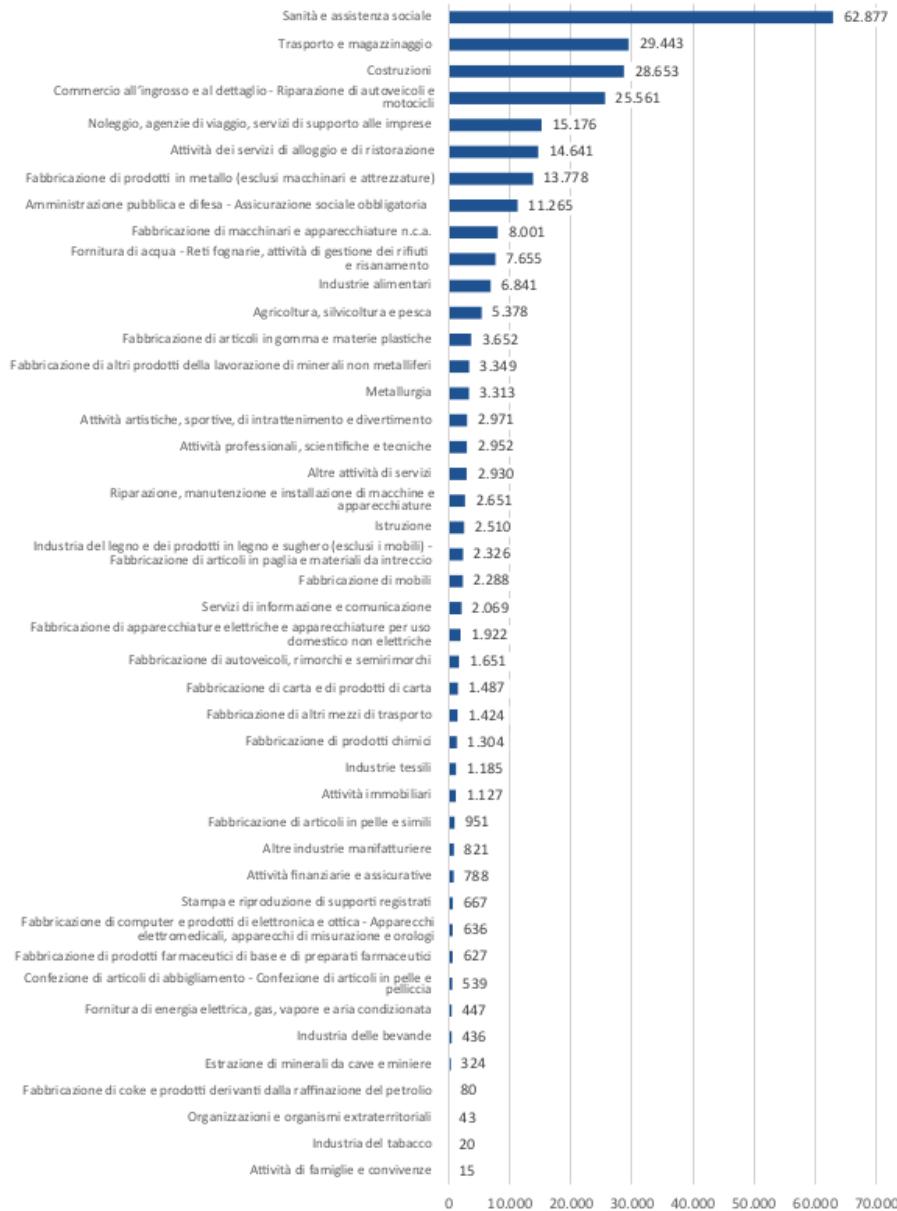


Figura 2.1: Infortuni sul lavoro accertati positivi per genere e modalità di accadimento nell'anno 2022.

Background



Fonte: Open data Inail. Data di rilevazione 30/04/2023

Figura 2.2: infortuni in occasione di lavoro accertati positivi per settore di attività nell'anno 2022

Essi comportano gravi conseguenze per i dipendenti, inclusi infortuni permanenti, invalidità e, nei casi più gravi, decessi. Oltre al costo umano, gli incidenti sul lavoro hanno un impatto significativo sull'economia delle aziende, generando costi diretti come spese mediche e indennità di infortunio, e costi indiretti come perdita di produttività, danni reputazionali e aumento dei premi assicurativi. L'EU-Occupational Safety and Health Administration (EU-OSHA) a questo proposito ha stimato in due diversi approcci l'impatto degli incidenti sul lavoro all'interno dell'Unione Europea^[2]. Nell'indagine sono stati presi in esame i dati relativi a 5 Paesi, poiché più completi e accessibili (tra cui figura anche l'Italia) e sono stati mostrati i risultati seguendo due diversi approcci: uno bottom-up, perché prende i valori dei costi per ciascun infortunio e li valuta globalmente; l'altro top-down, in quanto stima l'impatto dell'infortunio sulla vita del lavoratore e da valori macroeconomici come il PIL pro-capite valuta il costo effettivo dell'infortunio sul singolo. In termini pratici, nel primo caso si tiene conto dei costi diretti, indiretti e immateriali (effetti sulla vita e sulla salute) mentre nel secondo del valore monetario espresso in DALY, cioè il costo in termini di anni di vita persi a causa di un infortunio o di una malattia.

Paese	Finlandia	Germania	Paesi Bassi	Italia	Polonia
Numeri di casi	131 867	2 262 031	323 544	1 907 504	1 156 394
Costi diretti In Mio EUR	484	10 914	2 137	8 491	1 882
Costi diretti, % rispetto al totale	8	10	9	8	4
Costi indiretti In Mio EUR	4 362	70 658	6 468	58 961	19 588
Costi indiretti, % rispetto al totale	72	66	69	56	45
Costi immateriali In Mio EUR	1 196	25 557	5 147	37 392	22 311
Costi immateriali, % rispetto al totale	20	24	22	36	51
Onere economico complessivo In Mio EUR	6 042	107 129	23 751	104 844	43 781
Percentuale rispetto al PIL	2,9	3,5	3,5	6,3	10,2

Figura 2.3: onere economico complessivo stimato (approccio bottom up)

Il risultato di queste analisi ha mostrato che per l'Italia il costo di un infortunio o malattia causata dal posto di lavoro aveva un impatto percentuale sul PIL del 6,3% nel primo caso, mentre nell'approccio top down, riferendosi alla metodologia VSLY - considerata più coerente con i risultati dell'approccio bottom up - il valore mediano era del 7,7% rispetto alla produzione interna.

I valori ottenuti, indipendentemente dall'approccio utilizzato, non si discostano troppo l'uno dall'altro, confermando l'attendibilità dell'analisi. Si può dedurre perciò quanto questo problema sia concreto e impatti sulla società e sull'economia dell'Italia, dove il posto di lavoro è in gran parte costituito dall'industria.

	Germania	Finlandia	Italia	Paesi Bassi	Polonia
	DALY				
Totale dei DALY professionali	1 236 855	64 516	853 817	248 464	507 068
Percentuale rispetto ai DALY totali	4,9	4,2	5,1	5,7	4,0
DALY professionali per ogni 10.000 persone occupate	308	265	380	299	315
	Mio EUR	% rispetto al PIL	Mio EUR	% rispetto al PIL	Mio EUR
COSTI					
Approccio basato sul capitale umano					
Valore minimo	24 597	0,8	1 419	0,7	13 530
Media	55 429	1,8	3 106	1,5	31 475
Mediana	39 712	1,3	2 291	1,1	23 865
Massimo	138 404	4,5	7 393	3,5	69 671
Approccio WTP					
Valore minimo	32 324	1,1	1 637	0,8	20 929
Media	66 251	2,2	5 814	2,8	42 895
Mediana (")	66 251	2,2	4 335	2,1	42 895
Massimo	100 177	3,3	17 453	8,3	64 861
Approccio VSLY/VOLY					
Valore minimo	60 609	2,0	4 214	2,0	52 304
Media	191 939	6,3	9 345	4,5	133 789
Mediana	166 943	5,5	8 633	4,1	126 876
Massimo	420 489	13,8	19 425	9,3	256 120

Figura 2.4: stima dei costi complessivi approccio top down

L'utilizzo corretto dei Dispositivi di Protezione Individuale (DPI) è fondamentale per prevenire tali incidenti. Secondo la legislazione italiana, per DPI si intende *qualsiasi attrezzatura destinata ad essere indossata e tenuta dal lavoratore allo scopo di proteggerlo contro uno o più rischi suscettibili di minacciarne la sicurezza o la salute durante il lavoro, nonché ogni complemento o accessorio destinato a tale scopo*[3].

La normativa in materia di sicurezza sul lavoro è un sistema complesso e articolato, volto a tutelare la salute e l'incolumità dei lavoratori in ogni settore produttivo. Il fulcro di questo sistema è rappresentato dal **Decreto Legislativo 81/2008**, conosciuto come *Testo Unico sulla Salute e Sicurezza sul Lavoro*. Questo decreto introduce una serie di obblighi inderogabili per i datori di lavoro, al fine di garantire un ambiente salubre e sicuro. Tra i principi cardine si evidenziano:

- **Valutazione dei rischi:** il datore di lavoro, con l'ausilio di un responsabile di sicurezza ed un medico esperto, è tenuto ad effettuare un'attenta e completa valutazione di tutti i rischi presenti sul luogo di lavoro, compresi anche per gruppi di lavoratori specifici. A questo scopo deve redarre un documento dove vengono presi in considerazione tutti i criteri utilizzati nella valutazione dei rischi.
- **Programmazione della prevenzione:** sulla base della valutazione dei rischi, il datore di lavoro, nello stesso documento, deve individuare i dispositivi di sicurezza necessari nelle attività lavorative ed elaborare un piano di prevenzione, nell'ottica di eliminare o ridurre al minimo i rischi individuati. Questo piano deve essere integrato con le condizioni tecniche, ambientali e produttive dell'azienda, garantendo la sua effettiva applicabilità e sostenibilità.
- **Informazione e formazione dei lavoratori:** i lavoratori devono essere informati in modo chiaro e completo sui rischi generali dell'azienda e su quelli specifici a cui sono esposti durante lo svolgimento delle loro mansioni, su come effettuare un primo soccorso e a chi rivolgersi nell'ottica di prevenzione dei rischi. Devono inoltre ricevere una formazione adeguata sulla loro prevenzione, adottare comportamenti sicuri e utilizzare correttamente macchinari e dispositivi di protezione individuale. L'informazione e la formazione devono essere fornite prima dell'inizio dell'attività lavorativa e devono essere ripetute periodicamente, garantendo l'aggiornamento costante dei lavoratori, nel caso ad esempio vengano cambiate le mansioni, oppure siano introdotte nuove attrezzature e tecnologie.
- **Sorveglianza sanitaria:** questa misura è fondamentale per monitorare lo stato di salute degli operatori in relazione ai rischi cui sono esposti, prevenire l'insorgenza di malattie professionali e garantire l'idoneità alla mansione. La sorveglianza sanitaria è effettuata da un medico competente, che ha il compito di visitare i lavoratori, effettuare gli accertamenti sanitari necessari e rilasciare il giudizio di idoneità.

I DPI rappresentano l'ultima barriera di protezione, quando le misure tecniche e organizzative non sono sufficienti a eliminare o ridurre i rischi. Pertanto, la loro scelta, il loro utilizzo e la loro manutenzione devono essere effettuati con la massima attenzione e responsabilità. Vengono suddivisi nelle seguenti categorie in base alla loro funzione:

- **Protezione della testa:** caschi di protezione per l'industria.
- **Protezione dell'udito:** cuffie antirumore, tappi auricolari.
- **Protezione degli occhi e del viso:** occhiali protettivi, visiere.
- **Protezione delle vie respiratorie:** mascherine antipolvere e respiratorie.
- **Protezione degli arti superiori e inferiori:** guanti di protezione, scarpe antinfortunistiche, ginocchiere.
- **Indumenti di protezione:** tute, grembiuli, giubbotti ad alta visibilità.

Gli standard, giocano un ruolo fondamentale nel definire tecnicamente i criteri di produzione, utilizzo e manutenzione dei DPI, garantendo un elevato livello di protezione per gli utenti. La legge europea, come evidenziato nel **Regolamento (UE) 2016/425** stabilisce i requisiti che i DPI devono soddisfare nel mercato unico^[4], tra cui:

- **Ergonomia:** i DPI devono essere progettati e fabbricati in modo da essere comodi da indossare e non limitare la libertà di movimento del lavoratore, evitando di interferire con lo svolgimento delle sue attività, garantendone allo stesso tempo la sicurezza.
- **Livelli e classi di protezione:** i DPI devono fornire un livello di protezione adeguato al rischio specifico da cui proteggono. La classificazione dei DPI in base al livello di protezione consente di scegliere il dispositivo più idoneo in relazione al rischio da prevenire.
- **Marcatura:** i DPI devono essere marcati con il simbolo **CE**, a indicare la loro conformità ai requisiti di sicurezza dell'Unione Europea. La marcatura CE deve essere apposta in modo visibile, leggibile e indelebile sul DPI o sulla sua confezione.
- **Istruzioni e informazioni del fabbricante:** i DPI devono essere accompagnati da istruzioni chiare e complete (e.g. rischi coperti, prestazioni, classi di protezione, accessori, pezzi di ricambio etc.) per l'utilizzatore, che indichino in modo dettagliato come impiegare, conservare, pulire e manutenere correttamente il dispositivo. Le istruzioni devono

essere redatte in una lingua comprensibile nello Stato membro in cui il DPI è commercializzato. I dispositivi fabbricati devono avere una sorgente (il produttore e il suo indirizzo) ed essere identificati dal lotto messo in commercio.

2.2 Computer Vision

La computer vision è un campo dell'informatica incentrata sulla comprensione del contenuto di immagini o video per mezzo di un calcolatore. I task che si possono svolgere sono di diversi tipologie, tra cui la classificazione, l'object detection, la segmentazione, il riconoscimento di volti, l'encoding e l'applicazione di filtri per la modifica delle immagini originali. La ricerca sulle reti neurali nell'ambito della computer vision è stata tra le prime a mostrare le potenzialità di questa tecnologia nella risoluzione di problemi nel mondo reale. Storicamente l'insieme di diversi sviluppi nelle discipline di neuroscienza, deep learning e matematica ha permesso il raggiungimento di questo traguardo. Le scoperte relative al neurone biologico, la modellazione dei primi neuroni artificiali (assieme alla successiva estensione a più strati), l'utilizzo del calcolo differenziale per l'aggiornamento dei pesi ed infine la formulazione del teorema di approssimazione universale sono sicuramente gli elementi fondamentali di questo successo. Alla fine degli anni '50 è stato modellato il primo neurone artificiale, prendendo ispirazione dal neurone biologico, composto dalla combinazione lineare di input e pesi in ingresso ad una funzione di attivazione.

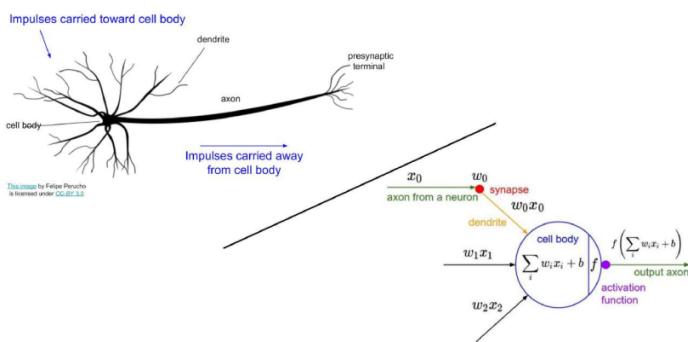


Figura 2.5: Modello del neurone artificiale sulla base del funzionamento di un neurone biologico.

Questo semplice meccanismo era in grado di mimarne grossolanamente il comportamento, generando una risposta a dei dati in ingresso, in modo tale che, superata una certa soglia, producesse o meno un valore in uscita. La funzione di attivazione era una semplice funzione gradino (al tempo non era scontato generare funzioni non lineari e continue), ma comunque questo oggetto era in grado di risolvere problemi di classificazione binaria. Il limite principale di questo modello consisteva nell'aggiornamento dei pesi in caso di predizioni sbagliate, basato su una delta di valori discreti. L'addestramento forniva in maniera euristica una direzione verso l'insieme ottimale delle variabili interne al modello, per ottenere la predizione il più possibile corretta ad ogni nuovo input.

Per risolvere questo limite, venne definita una funzione di attivazione continua, trasformando il problema da uno di classificazione ad uno di regressione. Questa nuova costruzione permetteva di introdurre una funzione di perdita, nell'ottica di minimizzare l'errore nelle predizioni attraverso un approccio più rigoroso. Dalla teoria delle regressioni lineari infatti si poteva utilizzare il metodo dei minimi quadrati, che in termini pratici significava ridurre il più possibile l'errore nella rappresentazione della funzione che si voleva apprendere dai dati. Fino alla fine degli anni '60 si sperimentò l'utilizzo di questi modelli, di cui gli esempi più famosi sono Adaline e Madaline, costituiti da semplici reti di neuroni artificiali, rispettivamente ad uno e due strati. Essi non riuscivano a rappresentare correttamente le non linearità all'interno della distribuzione dei dati, ma si trattava solo di un limite tecnico e non teorico, poiché non erano ancora state introdotte funzioni di attivazione non lineari continue come la sigmoide e non era ancora stato compreso come propagare l'aggiornamento dei pesi negli strati nascosti.

Nella seconda ondata di ricerca sulle reti neurali, iniziata negli '80, è stato dimostrato che è teoricamente possibile approssimare qualsiasi distribuzione dei dati attraverso l'apprendimento automatico di reti neurali con almeno uno strato di neuroni artificiali, aventi delle funzioni di attivazione non lineari. Questo teorema prende il nome di teorema di approssimazione universale. Esso si applica a tutte le tipologie più comuni di problemi risolti nel machine learning, quindi problemi discriminativi come la classificazione e la regressione e problemi generativi, come ad esempio l'encoding di immagini, la generazione di testo etc. Le implicazioni di questa dimostrazione hanno avuto un forte impatto solo in tempi più recenti, ma per comprenderne appieno le cause bisogna ancora revisionare alcuni elementi fondamentali in

questa storia. Dalla neuroscienza infatti, non si è soltanto preso ispirazione per la modellazione del perceptron, tant’è che a partire dagli anni ’50 è stato studiato il funzionamento della corteccia visiva nel cervello di alcuni mammiferi. Fondamentalmente con questi studi è stato dimostrato che i neuroni all’interno di questa zona sono organizzati gerarchicamente e nel livello più semplice rispondono a stimoli visivi con caratteristiche specifiche, come l’orientamento e le traslazioni.

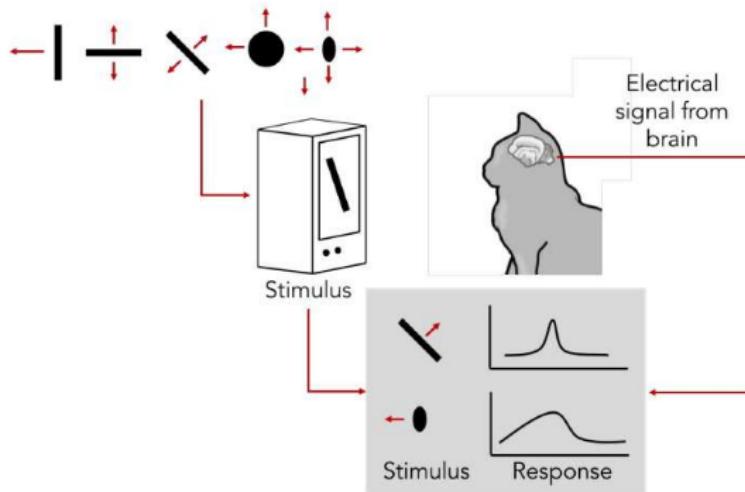


Figura 2.6: Gatto di Hubel e Wiesel[5]. Studi che hanno permesso di dare una definizione di recettore, scoprire l’organizzazione gerarchica della corteccia visiva nei mammiferi e formalizzare il concetto di retinotopia.

Nel 1980 venne proposto il Neocognitron, un antenato delle moderne reti convoluzionali. In questo modello sono stati trasposti i precedenti principi, in quanto, oltre ad implementare una architettura gerarchica con strati di neuroni, è stato definito matematicamente come modellare dei campi recettivi, cioè in che modo identificare delle forme semplici con diverse orientazioni dall’immagine di input, come succede per i recettori delle immagini provenienti dal campo visivo oculare. La definizione è stata presa della teoria dei segnali usando la formula della convoluzione:

$$y[n] = (x * h)[n] = \sum_{k=-\infty}^{+\infty} x[k] \cdot h[n-k] \quad (2.1)$$

Classicamente, questa espressione permette la generazione di diversi filtri, in modo tale da modulare o isolare solo parti del segnale di interesse, eliminandone altre che possono non essere utili a successive trasformazioni o semplicemente perché fonti di rumore. Nel dominio dell'Image processing si voleva sfruttare esattamente questa proprietà: applicare la funzione di convoluzione in modo da isolare le caratteristiche desiderate all'interno di una figura. Applicando questa formula nel dominio spaziale e definendo dei filtri bidimensionali, l'espressione assume la seguente forma:

$$y[i, j] = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} h[m, n] \cdot x[i - m, j - n] \quad (2.2)$$

Così non solo era possibile emulare il comportamento dei recettori visivi, ma allo stesso tempo si implementava il concetto di retinotopia. Il prodotto scalare di un filtro in una singola sezione dell'immagine, genera la stessa formula di un neurone artificiale, per cui ogni attivazione all'interno di ciascuna feature map (il risultato di una intera convoluzione) simula esattamente il modello del perceptron. La retinotopia definisce una relazione locale tra elementi vicini del campo visivo e neuroni vicini all'interno della corteccia visiva. Allo stesso modo attivazioni limitrofe nella feature map corrispondono ad elaborazioni di elementi prossimi nell'immagine. Nel complesso quindi, l'insieme di filtri e convoluzione permettono rispettivamente la rilevazione delle forme e la preservazione delle relazioni spaziali in un frame. Per ottenere invece un effetto di invarianza dalla posizione delle forme nell'immagine - in altri termini l'identificazione di queste indipendentemente da traslazioni - sono stati definiti degli strati di pooling. Questo modello presentava principalmente un grosso limite: il metodo di allenamento non era supervisionato e non si basava su una funzione di perdita globale, infatti l'utilizzo della backpropagation non era ancora stato formalizzato. Gli strati più interni della rete non permettevano la rappresentazione di forme più complesse e più coerenti con l'oggetto da classificare, come invece succede nelle reti moderne. Inoltre il Neocognitron era addestrato per il pattern recognition, ma non aveva una utilità pratica rispetto ai problemi più comuni nella computer vision. L'introduzione di uno strato fully connected, l'utilizzo di una funzione di perdita globale per la classificazione e della backpropagation portarono all'architettura di Lenet, nel 1998. L'allenamento di questa rete era specifico per la classificazione e tutti i neuroni dell'architettura partecipavano al training, quindi anche quelli degli strati

convoluzionali. In altre parole si trattava di una rete end-to-end. AlexNet, la rete che segna una netta linea di demarcazione nella storia del deep learning, mantiene la stessa architettura, con una principale differenza: le funzioni di attivazione all’interno della rete permettono la propagazione del gradiente senza saturazioni negli strati più interni.

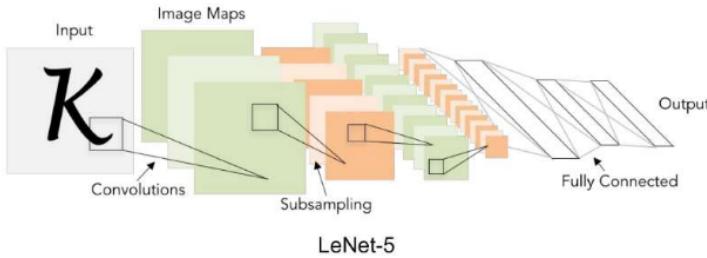


Figura 2.7: Lenet-5(1998). Primo modello ad aver dimostrato l’efficacia delle reti convoluzionali (CNN) nella comprensione delle immagini e ha aperto la strada a molte delle architetture moderne di deep learning. Crediti: Fei-Fei Li, CS231(Stanford).

La riduzione dell’errore nei problemi di classificazione nelle applicazioni di visione artificiale è stata solo una naturale conseguenza: l’architettura ormai era chiara e funzionante, si trattava solo di aumentare il numero di neuroni e strati all’interno della rete, grazie ad una potenza di calcolo che ai tempi di Lenet non era disponibile.

2.3 Cloud Computing nell’Industria

Il cloud computing rappresenta una delle innovazioni più rilevanti degli ultimi decenni nel settore IT, trasformando il modo in cui le aziende gestiscono le proprie risorse informatiche e processi produttivi. Questo nuovo paradigma, basato sull’erogazione di servizi tramite Internet, consente di accedere a risorse come server, storage, database e applicazioni software in modo scalabile e on-demand, senza dover effettuare investimenti iniziali significativi in infrastrutture hardware. Le implicazioni di questa trasformazione sono profonde, in quanto ridefiniscono i modelli di gestione IT e le strategie aziendali, favorendo un approccio più veloce e flessibile nell’implementazione di nuove soluzioni. La principale innovazione apportata dal cloud computing risiede

nella possibilità di adattare rapidamente le risorse informatiche alle necessità aziendali, garantendo una scalabilità notevolmente superiore rispetto alle infrastrutture tradizionali. In passato, le aziende che desideravano espandere i propri sistemi erano costrette a effettuare investimenti consistenti in hardware e a sostenere costi elevati per la relativa gestione e manutenzione. Inoltre, la diversa geolocalizzazione dei datacenter comporta vantaggi in termini di accessibilità, permette di risolvere problemi di latenza e di personalizzare i servizi in base alla regione in cui l'applicazione eseguita sul cloud viene deployata.

Oltre a facilitare la gestione delle risorse e ridurre i costi infrastrutturali, il cloud computing è considerato una tecnologia abilitante nell'implementazione dell'Industria 4.0. Ogni era industriale è stata segnata da una svolta tecnologica: nella prima l'introduzione della macchina a vapore, nella seconda l'elettrificazione delle macchine e la conseguente implementazione della catena di montaggio. La terza rivoluzione è stata possibile grazie all'invenzione del transistor e la successiva democratizzazione dei calcolatori. Questa nuova ondata invece è incentrata sui dati: ad esempio, nel 2015 è stato stimato che solo l'1% delle informazioni generate dai sensori all'interno di una fabbrica veniva effettivamente elaborata. Il trend tuttavia è destinato a cambiare, con un valore associato alle informazioni estratte dal mondo reale nell'ordine di 10^3 miliardi di dollari entro il 2025 [6]. L'adozione di tecnologie quali l'Internet of Things (IoT), gli sviluppi moderni nell'intelligenza artificiale e l'analisi di big data sono gli elementi che concorrono a questa nuova rivoluzione. Il primo di questi fattori è fondamentale per la generazione e l'ingestione, mentre gli altri due per il processamento: indipendentemente dalle loro funzioni, i dati restano il fulcro delle operazioni. In questo contesto, il cloud fornisce l'infrastruttura e i servizi necessari per l'integrazione di questi elementi, rendendo possibile l'implementazione delle smart factories.

La capacità del cloud di raccogliere, archiviare ed elaborare grandi quantità di dati in tempo reale è cruciale per sfruttare appieno il potenziale dell'Industria 4.0. Le aziende che operano in settori industriali tradizionali, come la manifattura, possono trasformare le loro linee di produzione in sistemi autonomi e ottimizzati, capaci di adattarsi alle esigenze del mercato e di ridurre significativamente gli sprechi. La connettività fornita dal cloud consente invece di collegare dispositivi, sensori e macchinari all'interno della fabbrica, creando un ecosistema in cui ogni componente è in grado di comunicare e condividere le proprie informazioni, rendendo più semplice il monitoraggio dei processi produttivi. Inoltre con gli avanzamenti nella ricerca sul

deep learning, diventato sempre più consistente negli anni, i relativi modelli sono stati adottati per migliorare diverse parti dell’ecosistema aziendale, come il monitoraggio, i processi decisionali e produttivi. Un esempio concreto è la manutenzione predittiva, che sfrutta i dati provenienti dai sensori per rilevare anomalie e prevedere i guasti delle macchine. E’ così possibile ridurre i tempi di inattività, prolungare la vita utile delle apparecchiature e migliorare la loro efficienza complessiva. Sempre nello stesso contesto, un’azienda potrebbe utilizzare il cloud per raccogliere e analizzare dati provenienti dalle linee di assemblaggio, applicando modelli di apprendimento automatico per migliorare la qualità dei componenti e ridurre i difetti di produzione. La sicurezza in fabbrica, oggetto di questa trattazione, rientra chiaramente nel dominio delle applicazioni in questa fase industriale.

2.4 Lavori Correlati

In letteratura, sono stati individuati due approcci che rientrano nell’ambito di questa ricerca. Nel complesso, dimostrano delle differenze nella modalità di implementazione del sistema, rispetto a quanto realizzato in questa tesi, ma aiutano a fornire contesto alla trattazione. Il primo metodo [7] è totalmente integrato con il cloud, ma si serve del provider Microsoft Azure. L’obiettivo del paper è quello di analizzare direttamente su una telecamera IoT, in real time, l’input proveniente da una Learning Factory, un ambiente che simula quello di un impianto manifatturiero. Si tratta di un luogo pensato a scopo di training e di ricerca, ma non per questo esente da rischi. Il modello eseguito sul SoC del dispositivo è stato allenato e customizzato per essere lanciato su dell’hardware con una potenza di calcolo limitata. La fase di generazione del modello avviene sul cloud, successivamente, una versione leggera viene caricata sul dispositivo. Come nel sistema implementato in questa tesi, l’architettura è di tipo edge-cloud. Le differenze sostanziali consistono nella locazione del modello e nella modalità in cui esso viene acceduto. Non c’è bisogno infatti di invocazioni a servizi nel cloud una volta che l’algoritmo di machine learning è stato deployato sull’edge, rispetto a quanto succede invece nel sistema proposto. In particolare, l’implementazione oggetto di questa tesi, chiama una API per un modello già specializzato nell’analisi dei dispositivi di sicurezza, come si vedrà nel capitolo successivo. Nel related work, inoltre, l’obiettivo è l’identificazione degli occhiali protettivi, mentre il modello utilizzato in questa soluzione è in grado di rilevare

ulteriori classi di dispositivi di sicurezza. L'algoritmo deployato sull'edge raggiunge l'obiettivo di inferenza in tempo reale, ma le prestazioni sono comunque limitate, indipendentemente dal tipo di allenamento eseguito. A questo proposito, sono stati seguiti due approcci: nel primo caso il training è effettuato su immagini presenti in rete, di fatto poco adattibile allo scenario della Learning Factory. Nel secondo invece, sono state utilizzate immagini provenienti dal sito, ottenendo un benchmark migliore, seppur limitato.

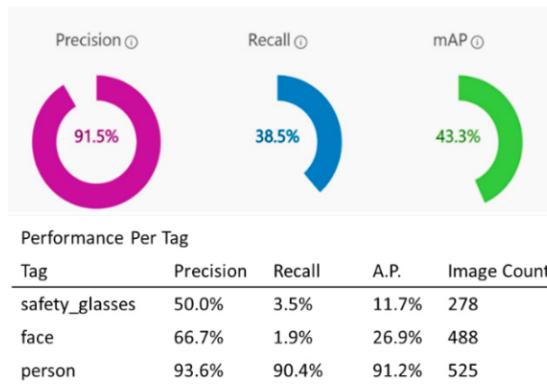


Figura 2.8: Prestazioni Learning Factory: custom dataset.

Il modello rileva tre classi principali: occhiali protettivi, volti e persone. Gli ultimi due tag sono importanti per capire se i dispositivi sono indossati, mentre il primo serve per l'identificazione dell'oggetto di interesse. I risultati confermano la difficoltà generale dei modelli nel trovare elementi piccoli all'interno di un frame. Sul totale dei casi in cui il modello è in grado di identificare un oggetto, solo la metà sono occhiali protettivi. In generale l'algoritmo riesce ad identificare pochissime volte questo tipo di dispositivo, proprio per le loro piccole dimensioni.

Il secondo approccio analizzato [8], propone una soluzione completamente on premise. Si tratta di un meccanismo molto più sofisticato rispetto all'oggetto di questo scritto, ma, trattandosi di un lavoro recente e con prestazioni ottimali, permette di ottenere una panoramica migliore sugli sviluppi in questo campo. Il sistema proposto non si limita solo a rilevare i dispositivi di sicurezza, ma anche gli incidenti e il controllo delle macchine. In base a ciò che il modello utilizzato riesce ad identificare, viene generata una matrice composta dalla probabilità che un evento dannoso possa verificarsi e dalle classi di gravità dell'infortunio. Il sistema, al matching sulla matrice, deve

rispondere opportunamente. Questo meccanismo introduce un controllo della granularità nella reazione agli eventi, in quanto, ad esempio non ha senso spegnere un macchinario se un lavoratore non ha un dispositivo di sicurezza indossato, ma allo stesso non si trova in un'area pericolosa.



Figura 2.9: Sistema per la gestione della sicurezza in fabbrica.

La ricerca si basa sempre su modelli di apprendimento automatico per la computer vision: si tratta di una soluzione sull'edge, ma non a livello di telecamere. Non c'è una integrazione con il cloud, infatti l'algoritmo, una Faster R-CNN, viene addestrato in Colab e successivamente deployato su un server all'interno della fabbrica. Questa tipologia di rete neurale convoluzionale è un ottimo trade-off in termini di utilizzo delle risorse, di accuratezza nelle rilevazioni e di velocità di esecuzione. Vengono mostrate le metriche su diverse Intersection of Union (IoU) per le classi di DPI in esame, cioè casco protettivo e giubbotti di sicurezza. Anche in questo studio è stato utilizzato un dataset ad hoc per migliorare l'accuratezza nella rilevazione. I risultati ottenuti, raggiungono lo stato dell'arte per il dominio di applicazione, con una mean average precision di 0,74 e una mean average recall dello 0,83.

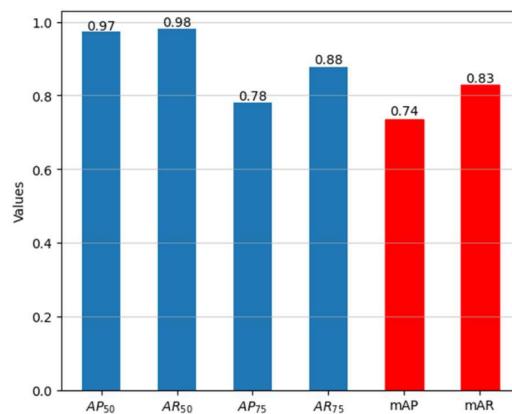


Figura 2.10: Risultati del sistema di rilevazione.

Capitolo 3

Tecnologie e Servizi

3.1 Docker

Docker è diventato negli anni il sistema più popolare per gestire la virtualizzazione basata su container. Può essere visto come un tipo di macchina virtuale leggera, in quanto riesce a garantire le stesse caratteristiche di quelle tradizionali, ma con un overhead minore, sia in termini di efficienza nell'esecuzione, che di spazio occupato.

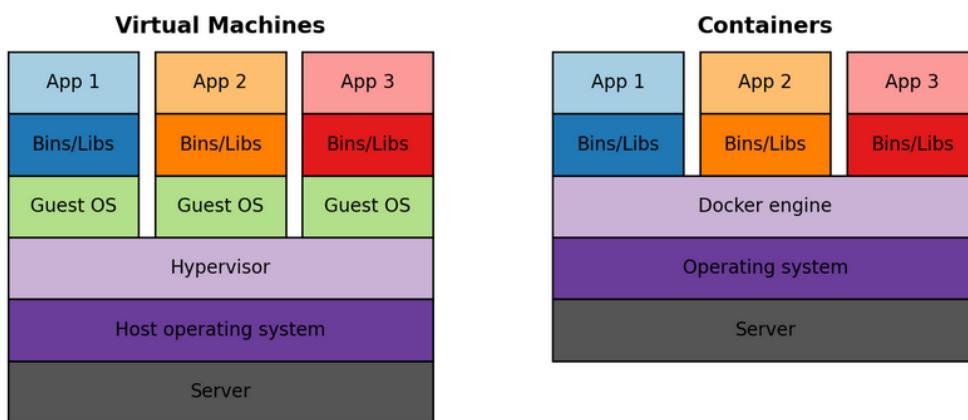


Figura 3.1: Confronto tra macchine virtuali tradizionali e containers.

Essi vengono eseguiti direttamente sul sistema operativo dell'host, senza la necessità di chiamate di sistema multiple. Vengono impacchettate solo le librerie necessarie nell'immagine di base fornita al container. In realtà, la

sua value proposition è legata alla possibilità di poter lanciare una applicazione ovunque, con la garanzia che questa funzioni sempre. L'analogia con i container nel mondo reale è diretta: quelli utilizzati nel commercio, possono essere adattati indipendentemente dall'ambiente di trasporto. Allo stesso modo l'applicazione che viene eseguita, funzionerà indipendentemente dall'ambiente di esecuzione, qualsiasi sistema operativo esso sia, con l'unica limitazione sull'instruction set per cui è stata generata la build. Nella pratica, quindi, esistono due definizioni in base all'utilizzatore di questa tecnologia: se si parla di gestione di una infrastruttura, allora l'attenzione è spostata verso la virtualizzazione, mentre se il focus è relativo alle applicazioni, allora si può vedere Docker come una tecnologia orientata alla portabilità. In generale, le macchine virtuali devono garantire un serie di proprietà per poter essere definite tali, che sono:

- **consolidamento:** inizialmente ciascuna applicazione veniva eseguita su una macchina distinta, rendendo l'utilizzo delle risorse inefficiente. La virtualizzazione ha permesso di superare questa limitazione, con l'esecuzione di software differenti all'interno della stessa macchina.
- **isolamento:** questa proprietà consente ad ogni applicazione di vedere l'ambiente come se fosse una macchina a sé stante, nonostante il consolidamento. I sistemi operativi offrono funzionalità di isolamento base come l'astrazione dello spazio di indirizzamento di un processo, ma questo non è sufficiente per una applicazione self-contained. Diversi applicativi infatti possono interferire tra di loro già solo nell'utilizzo della memoria, nel caso di sovra-allocazioni, portando ad un degrado delle prestazioni o al crash(verifica). Lo stesso ragionamento si può applicare anche per la gestione della cpu, della rete e del filesystem che all'interno di un sistema operativo sono risorse completamente condivise tra processi.
- **flessibilità:** il controllo dei container è molto semplice in quanto docker prevede dei comandi standard per la loro gestione, come l'avvio, la pausa e lo stop. Gli orchestratori giocano un ruolo fondamentale per riallocare i container in nodi diversi dell'infrastruttura, per esempio in caso di manutenzione o sovraccarico. Questo processo avviene senza particolari overhead, in quanto i container, a differenza delle macchine virtuali tradizionali hanno dei tempi di attivazione molto bassi.

- **portabilità:** come già visto, i container possono essere avviati in qualsiasi piattaforma, indipendentemente dal sistema operativo, dall'hardware, dalle librerie necessarie e dalle loro versioni. Questo comporta un grande vantaggio anche nell'uso di linguaggi di programmazione di basso livello quando l'efficienza è fondamentale. Linguaggi come Java, ad esempio, si servono di macchine virtuali per poter garantire la portabilità del codice, ma il tempo di esecuzione aumenta notevolmente.

L'implementazione dei container è stata possibile grazie a due elementi presenti all'interno dei sistemi operativi: namespaces e control groups(cgroups). Si tratta di features che inizialmente non erano state pensate per la virtualizzazione, ma solo nell'ottica di avere dei processi isolati. Non si è quindi arrivati subito alla definizione di container, ma ci sono state delle tappe risolutive di diversi problemi che gli sviluppatori avevano all'inizio degli anni 2000. I namespaces permettevano la gestione della visibilità relativa alle risorse per ogni processo, mentre i cgroups servivano a limitare l'utilizzo di quanto allocato. Nel secondo caso, esistevano già dei meccanismi di questo tipo all'interno del kernel Linux (e.g. cpulimit, nice), ma non erano centralizzati e potevano essere applicati solo a singoli processi. L'evoluzione dei cgroups è fondamentalmente quella di estendere il tutto ad un gruppo di task in maniera modulare, con la possibilità di definire delle gerarchie nella gestione delle risorse. Le tecnologie basate su container come Docker hanno integrato questi meccanismi nel loro layer di virtualizzazione, in modo tale da fornire delle primitive di alto livello per la gestione dei container, che altrimenti con le precedenti astrazioni sarebbero accessibili solo ad utenti esperti, ed in ogni caso inclini ad errori a causa della complessità di utilizzo.

Docker presenta infine una particolarità a livello di filesystem: rispetto alle macchine virtuali tradizionali o container LXC, che sfruttano un filesystem in senso stretto, questa tecnologia utilizza il concetto di Union Filesystem.

Si tratta di un sistema che combina diversi filesystem restituendo logicamente un'unica struttura. Nell'implementazione di Docker, lo Union Filesystem si presenta come una gerarchia di strati, tutti in sola lettura, a cui ne viene aggiunto uno scrivibile al lancio del container. Questo meccanismo, come si può vedere in Figura 3.2 permette la condivisione della stessa immagine di base, evitando allo stesso tempo di dover duplicare lo spazio di memoria e che modifiche da parte di una macchina possano influenzare ciò che vedono gli altri container basati sulla stessa istanza, attuando così

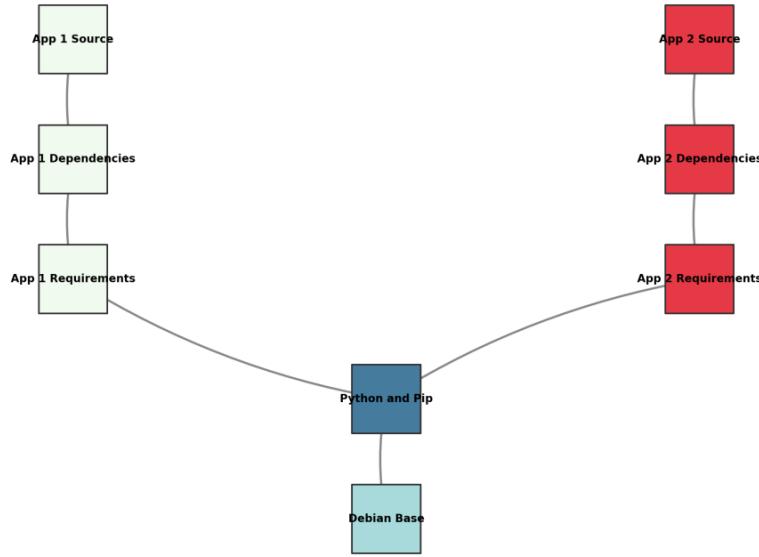


Figura 3.2: Esempio di condivisione di layers in Docker. Le due immagini risultanti hanno lo stesso sistema di base, condividendo lo stesso ambiente (Debian) e runtime (Python).

un primo livello di isolamento (quello più generale con l'host viene sempre implementato attraverso i namespaces).

La realizzazione pratica di questa funzionalità è possibile grazie al concetto di copy-on-write, già utilizzato nella creazione dei processi. In questo caso il funzionamento è analogo, ma si applica a livello di memoria secondaria. Ogni modifica all'immagine di base, attraverso il layer in scrittura di più alto livello, è ciò che viene salvato. Non si apportano modifiche all'intera memoria: in pratica è un meccanismo basato sulle differenze. Una volta che si esegue il commit di un'immagine, gli strati vengono congelati e ne formano una nuova. Al livello dell'utilizzatore questo sistema si rivela molto utile perché si possono comporre nuove immagini impilando nuovi layer a quelli già esistenti, senza preoccuparsi dei livelli sottostanti, perché grazie alla portabilità il loro funzionamento è garantito. La composizione non è l'unica proprietà interessante che emerge da questa architettura, ma anche modularità, riutilizzo ed efficienza. Quest'ultima è importante sia in termini di spazio, perché come già visto i layer inferiori vengono condivisi, sia in termini di tempo. Quando si costruisce una nuova immagine, infatti, verranno aggiunti o scaricati solo gli strati di più alto livello per la generazione della nuova immagine, invece di dover ri-eseguire l'operazione ogni volta.

3.2 MQTT

Message Queue Telemetry Transport (MQTT) è un protocollo orientato ai messaggi, basato su TCP, il cui obiettivo è quello di fornire un protocollo leggero, per dispositivi con risorse limitate e connessioni instabili. Il formato dei dati infatti è semplice ed è pensato per essere affidabile e fault-tolerant. Proprio per le sue caratteristiche, con il tempo è diventato uno standard nelle applicazioni IoT, ad esempio monitoraggio remoto e raccolta di dati in industria, domotica e ambiente. MQTT si basa su un’architettura publish-subscribe. Ciò permette di disaccoppiare l’invio dei messaggi tra produttori e consumatori, attraverso un broker. Si tratta di un metodo di comunicazione indiretto, in quanto per chi pubblica non è necessario conoscere la destinazione, come invece avviene nelle architetture client-server. Il funzionamento è analogo a quello delle newsletter. Quando un utente è interessato ad un certo argomento, lascia il proprio indirizzo email, in modo tale da ricevere aggiornamenti. Analogamente in un sistema publish-subscribe le entità che devono ricevere determinati messaggi da un publisher, si iscrivono ad un canale.

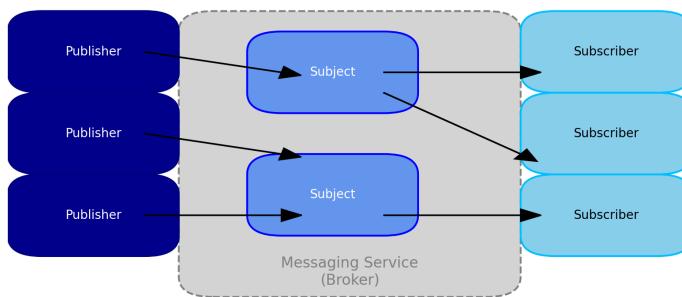


Figura 3.3: Schema architettura publish-subscribe.

Oltre al disaccoppiamento, questo tipo di pattern fornisce ulteriori vantaggi[9]. È possibile aggiungere nuovi publisher o subscriber senza dover modificare la logica esistente, il che rende l’architettura facilmente scalabile. Si tratta di uno dei motivi per cui il cloud riesce a gestire facilmente milioni di messaggi in un breve intervallo di tempo ed avere migliaia di publishers.

Inoltre ogni parte del sistema non dipende dalle altre, favorendo la comunicazione asincrona, poiché nessun publisher deve aspettare per l'invio di nuovi messaggi prima che i consumatori finiscano. Allo stesso modo i consumatori non devono attendere che altri ricevano i messaggi per poter proseguire con l'elaborazione delle proprie informazioni. Questo tipo di pattern viene implementato in tanti sistemi come quelli delle notifiche push nelle applicazioni mobili, nei servizi gestiti del cloud ed infine nei sistemi IoT, attraverso il protocollo MQTT, focus dei paragrafi successivi. In questo contesto, le unità di interesse vengono descritte come topic. Essi sono univocamente definiti all'interno dei broker attraverso una struttura gerarchica che ricorda i path nelle URI. Ad esempio, un topic può essere dichiarato come: `factory/floor/sensor1`. Non esiste un meccanismo esplicito di creazione dei topic, ma vengono definiti contestualmente alla pubblicazione di un messaggio.

Il broker ha la responsabilità di connettere, se autorizzati, i diversi client della rete, di tenere traccia dello stato di connessioni precedenti e di filtrare i dati in arrivo dai publisher. Per il discorso di efficienza previsto da questo protocollo, infatti, devono essere distribuiti solo i pacchetti necessari al funzionamento del sistema, e quindi la capacità di filtro è un elemento fondamentale. In particolare, esistono diverse tipologie di controllo dei pacchetti in transito. Quello più importante è il topic filter, dove un messaggio deve contenere necessariamente un topic nell'header, e sarà compito del broker ritrasmetterlo o ignorarlo in base a come è stata implementata la sua logica. In generale i filtri vengono usati per riferirsi ad un insieme di topic, sfruttando due tipologie di wildcards. Si utilizza il segno "+" per indicare un intero livello nella gerarchia del path definito dal topic. Ad esempio `factory/+/sensor1` serve per tutti i sensor1 nei diversi piani di una fabbrica. Il secondo simbolo, cioè "#" si riferisce ad una intera sottogerarchia nel path. `factory/#` e `factory/first/#` filtrano rispettivamente tutti i messaggi provenienti dalla fabbrica e quelli provenienti dal primo piano di un complesso industriale.

MQTT è un protocollo progettato con l'affidabilità in mente: in base al contesto (hardware, banda, latenza) e al tipo di applicazione, fornisce diversi livelli di Quality of Service(QoS):

- **QoS0:** modalità di funzionamento default in MQTT, se non specificato. I messaggi vengono inviati con una logica fire-and-forget, senza la garanzia che il processo vada a buon fine. Il publisher o il broker inviano

i messaggi senza alcun tipo di risposta da parte di un subscriber. Non ci sono ulteriori garanzie oltre a quelle fornite dal protocollo TCP.

- **QoS1:** garantisce che il messaggio venga inviato almeno una volta. Questo meccanismo si basa su messaggi di acknowledgment provenienti dal ricevente, per cui se non arriva nessuna risposta, il messaggio deve essere ritrasmesso.
- **QoS2:** si tratta del livello di qualità più alto. Il messaggio viene inviato una sola volta, grazie ad una serie di messaggi di controllo generati dalle entità in gioco. Può essere visto come un acknowledgement incrociato tra publisher e ricevente. Quello che avviene in più rispetto a QoS1 è l'invio dei pacchetti publish-release per indicare il rilascio del messaggio dalla memoria del publisher, e publish-complete da parte del ricevente per chiudere la comunicazione. In questo modo alla fine del processo il messaggio è stato inviato e gli attori in gioco sono totalmente sincronizzati dal punto di vista della ricezione.

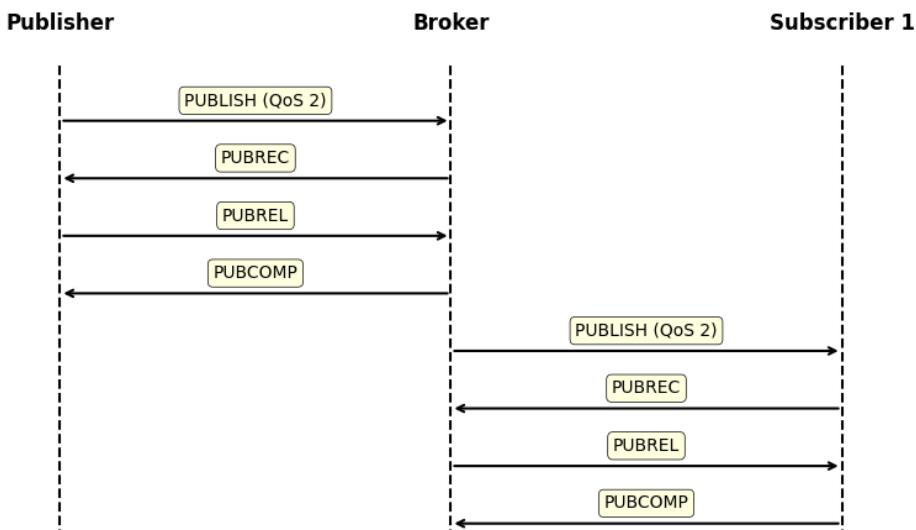


Figura 3.4: Flusso pacchetti di controllo per QoS2. I livelli più semplici funzionano in maniera simile, con un numero di pacchetti inferiore. Si può notare come il disacoppiaamento fornito dal broker faciliti lo scambio dei messaggi.

Il QoS viene negoziato al momento della connessione con il broker, ed in base alle sue capacità può essere dinamicamente modificato in un secondo

momento. Nell'affrontare il concetto di qualità di servizio, si è visto quindi il funzionamento della publish da parte di un client, ma esistono altri tipi di operazioni in questo pattern, cioè subscribe, unsubscribe e ping. In particolare quest'ultimo risulta fondamentale per l'implementazione del Last Will and Testament(LWT), un metodo per comunicare la disconnessione di un publisher all'interno della topologia. LWT è un messaggio contenente il topic, il QoS e un payload da inoltrare a tutti i subscriber che fino a quel momento ricevevano le notifiche. Questo pacchetto viene fornito nel momento in cui un client si connette per la prima volta al broker, prima che inizi l'invio effettivo dei dati per gli altri client. Se attraverso delle operazioni di ping, impostate con un certo timer, una delle due parti non riceve risposta, allora si assume esserci stata una disconnessione. A quel punto, se è il client ad essere offline, il broker invierà il Will ai subscribers. Un'ultima caratteristica importante per il funzionamento di MQTT, è la persistenza. Il broker mantiene le informazioni di sessione per i client disconnessi, in modo tale da riottenere facilmente le iscrizioni ai topic e i messaggi in coda che non avevano ricevuto (se si tratta di QoS1 e QoS2, pensati come già visto per la trasmissione affidabile). Inoltre, i publisher possono impostare un flag RETAINED nei messaggi, in modo tale che vengano salvati in maniera indefinita nel broker. L'obiettivo di questa funzionalità è garantire l'invio di un determinato messaggio ad un client indipendentemente dal momento della sua iscrizione al topic.

3.3 Protocolli di streaming locale

La rete IP è nata per la trasmissione di dati in zone geograficamente distanti tra loro. Le comunicazioni inizialmente erano basate sull'invio di file ed e-mail. La naturale estensione di questo meccanismo è stata quella di poter trasferire informazioni multimediali come animazioni, voce e video in tempo reale, il che ha comportato l'introduzione di nuovi requisiti nello stack IP. Se da un lato il protocollo di livello 3 era sufficiente per le comunicazioni più semplici, esso risultava inaffidabile sia in termini di mantenimento di informazioni, che di ordine di arrivo. L'introduzione del livello TCP ha risolto questa limitazione, ma con l'avvento dello streaming si sono presentati nuovi problemi, legati soprattutto alla latenza e al jitter, rispettivamente il tempo totale per la trasmissione di un pacchetto e la variabilità del delay.

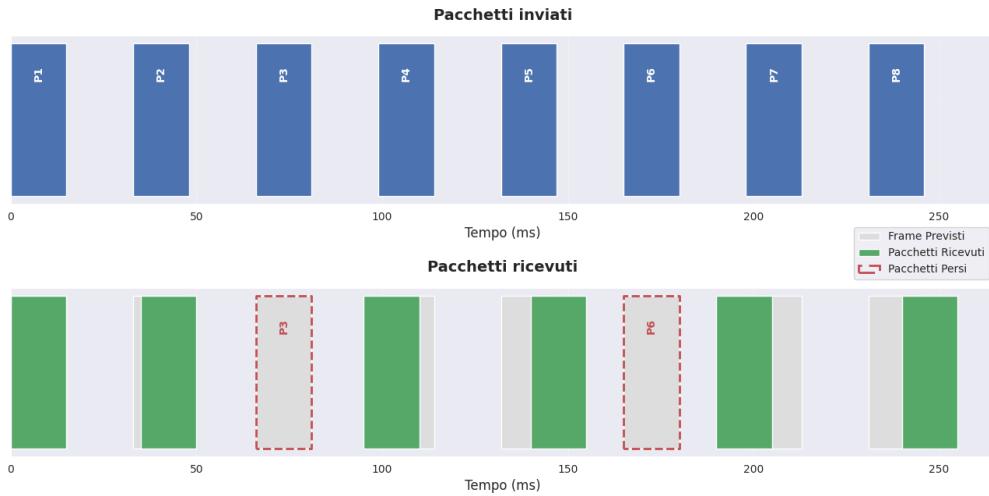


Figura 3.5: Effetto del jitter nella riproduzione dello streaming. Senza buffering i pacchetti non vengono riprodotti seguendo la stessa temporizzazione della sorgente.

Esistono 2 principali tipologie di software in questo contesto: interattivi (video-conferenze) e passivi (live-streaming). Nel primo caso i requisiti sul ritardo e sul jitter sono molto stringenti, per cui non si possono introdurre meccanismi pesanti di bufferizzazione, poiché compromettono irrimediabilmente la user experience. Possono invece essere implementati nel secondo, in modo tale da mitigare gli effetti della trasmissione. Bisogna ridurre al minimo i tempi di attesa: ad esempio il reinvio dei pacchetti, come succede in TCP, non può essere implementato. UDP invece si presta a questo meccanismo ed infatti fornisce la base per un protocollo di streaming in tempo reale. Non instaura nessun tipo di connessione e non deve monitorare le condizioni della rete per adattare la sua velocità nella comunicazione. Real-time Transport Protocol(RTP) è un protocollo di livello trasporto[10]è stato progettato per lavorare sopra UDP, e con le sue caratteristiche è in grado di soddisfare i requisiti per la trasmissione di dati multimediali su internet. RTP infatti permette la negoziazione di una codifica comune ai lati della comunicazione. Grazie a questo protocollo, i livelli applicativi sono liberi di implementare il buffering in base alle proprie necessità, poiché integra nei pacchetti il timestamp del contenuto multimediale. RTP non è sensibile al livello di congestione della rete, quindi il rate di trasmissione non viene modificato, come invece succede in TCP.

RTP lavora in coppia con Real-time Control Protocol (RTCP), la sua controparte per la verifica dello stato della rete. A differenza di TCP, questo protocollo ha una funzione di solo monitoraggio, non serve per reagire alle congestioni, quindi ai ritardi, alle perdite o al jitter. Si tratta di un protocollo leggero, che periodicamente invia messaggi di controllo verificando lo stato della rete, in modo tale che i livelli applicativi, in base alla qualità che vogliono fornire, reagiscano di conseguenza. Si è delegato quindi al software questo onere, senza fornire una maniera standard per il controllo del traffico. Questo è un grosso vantaggio in termini di flessibilità: sfruttando TCP al contrario, la velocità di trasmissione dipenderebbe dalle sue regole nella gestione del flusso di informazioni. Esiste infine un protocollo di più alto livello per la gestione delle sessioni nella trasmissione dati: Real-time Streaming Protocol (RTSP). Esso non invia i pacchetti effettivi per lo streaming, perché svolto interamente da RTP, ma una volta che la sessione è attiva, serve per inviare i comandi relativi allo stato di quest'ultima. Per cui il client che richiede dati multimediali manderà al server dei messaggi per avviare, sospendere o fermare il flusso di informazioni.

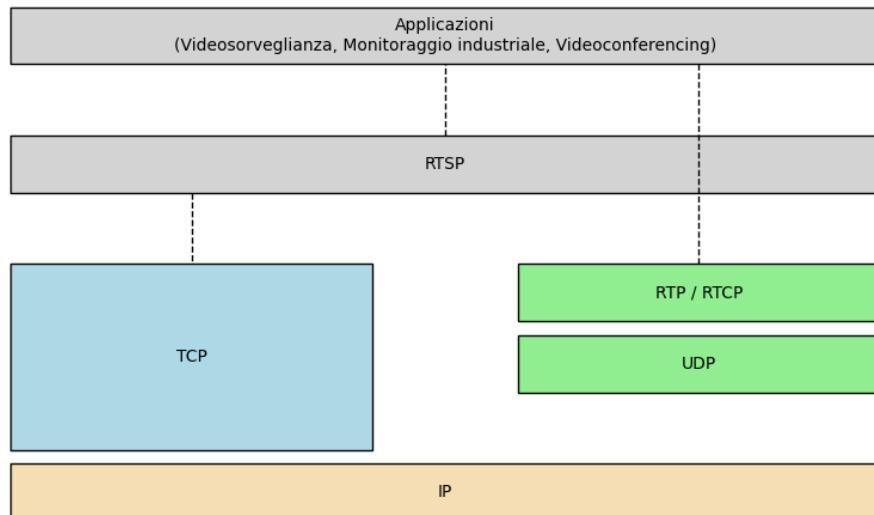


Figura 3.6: Stack di protocolli per lo streaming real-time.

Il formato dell'header RTP è costituito da tre elementi principali. I primi 12 bytes sono sempre presenti e formano il core del pacchetto. Successivamente possono esserci delle estensioni (anche se usate molto raramente) ed infine c'è il campo payload, il cui formato è deciso dall'applicazione. Nel

dettaglio i campi di RTP sono:

- **version**: bit per l'identificazione della versione di RTP usata nella trasmissione.
- **padding**: se questo campo è configurato, sarà presente del padding nel pacchetto, ad esempio per allineamento oppure se si sta utilizzando qualche algoritmo di cifratura.
- **extension**: quando il bit è attivo, il campo extention header sarà presente nel pacchetto.
- **CSRC Count**: conta il numero di sorgenti multiple, quando sono presenti, come spiegato nel campo CSRC.
- **marker**: identifica punti specifici nel flusso multimediale, come i confini tra diversi frame in un video.
- **payload type**: campo per la decodifica del contenuto multimediale nel payload.
- **sequence number**: numero identificativo del pacchetto. E' utile per stabilire se ci sia stata una perdita e permettere alle applicazioni di reagire. Queste possono decidere di non procedere in nessun modo oppure di attuare delle strategie. Ad esempio si potrebbe utilizzare delle codifiche che richiedono meno banda o riprodurre il frame precedente/successivo più volte.
- **timestamp**: indica il momento di generazione del pacchetto attraverso il clock di sistema. Serve per sincronizzare i flussi audio e video. Viene inoltre utilizzato per determinare il jitter.
- **Synchronization Source Identifier(SSRC)**: identifica la sorgente nella comunicazione RTP. E' un valore scelto in randomicamente a livello locale dalle fonti, quindi esiste la possibilità di collisioni. In tal caso RTCP non permette che questo evento si verifichi, facendo uscire e rientrare dalla sessione una delle due sorgenti.
- **Contributing Source Identifiers(CSRC)**: è una lista delle sorgenti che contribuiscono al payload del pacchetto, anche se in generale il source è solitamente uno solo. E' utile nel caso delle videoconferenze

quando più fonti generano un stream di dati, ma si vuole compattare i flussi in un unico pacchetto per necessità di banda.

In RTP non è presente un concetto esplicito di sessione, ma esso viene definito contestualmente alla generazione dello streaming, identificando il destinatario tramite indirizzo IP e una coppia di porte RTP ed RTCP.

RTSP è nato come standard per la definizione delle sessioni multimediali ed il loro controllo. In RFC2326 è definito come "remote control" nella gestione dello streaming, perché si presenta con una serie di comandi, che ricordano molto quelli di HTTP. Tuttavia differisce da questo protocollo in un punto fondamentale: RTSP è simmetrico, in quanto sia il client che il server multimediale possono inviare richieste. Inoltre i messaggi non sono di tipo fire and forget, infatti i comandi inviati possono riferirsi a richieste precedenti, rendendo di fatto RTSP un protocollo stateful[11]. Non è un meccanismo orientato alle connessioni, per cui anche appoggiandosi a TCP potrebbe più volte aprire diverse comunicazioni attraverso il protocollo di livello trasporto nella stessa sessione RTSP. In generale il suo funzionamento non dipende dal protocollo di livello 4 sottostante, per cui si può appoggiare sia a TCP che ad UDP nella generazione dei propri messaggi, ma tipicamente viene utilizzato TCP. Nonostante i requisiti di bassa latenza per l'intero sistema di streaming, nel caso di RTSP è importante che i comandi arrivino a destinazione, per cui si utilizza un canale di comunicazione affidabile, separato dal flusso vero e proprio. Questo non comporta degrado delle prestazioni perché i pacchetti sono su un canale separato e la quantità è talmente limitata che non impatta sulla banda disponibile. Infine, per quanto riguarda le sue caratteristiche, bisogna menzionare come vengono identificati gli stream. Si sfrutta un RTSP URL (e.g. `rtsp://hostname/video/front`), in maniera molto simile a come vengono definite le risorse in HTTP. Questo mapping è molto utile perché può applicare i comandi a più stream contemporaneamente, seguendo la struttura gerarchica dei path nell'URL. Ad esempio se ci sono più flussi video, come `rtsp://hostname/video/front` e `rtsp://hostname/video/top`, inviando un comando a `rtsp://hostname/factory/video`, questo si applicherà ad entrambe le risorse. I metodi disponibili in RTSP sono:

- **OPTIONS:** insieme dei comandi che entrambe le parti possono accettare, serve quindi per verificare la compatibilità delle operazioni disponibili tra gli estremi di comunicazione.

- **DESCRIBE**: questo comando fornisce la descrizione del contenuto multimediale come il formato e la codifica.
- **SETUP**: inizializza la sessione di streaming RTSP, tramite dei parametri (e.g. porta, protocollo di trasporto) dopo che entrambe le parti hanno allocato le risorse necessarie.
- **PLAY**: il client invia questo comando quando richiede l'effettiva trasmissione del flusso.
- **PAUSE**: lo streaming viene interrotto dal client senza che il server liberi le risorse e i parametri di sessione.
- **TEARDOWN**: il client richiede lo stop dello streaming, seguito dalla liberazione delle risorse nel server.
- **GET_PARAMETER, SET_PARAMETER**: questa serie di comandi fornisce o imposta i parametri di sessione nello streaming (ad esempio i pacchetti ricevuti, il jitter etc).

RTSP definisce una macchina a stati, che permette la coordinazione del flusso di streaming multimediale, in modo tale da consentire ai partecipanti della sessione di sapere quali comandi possono inviare o ricevere in ogni sua fase. Sia il client che il server memorizzano lo stato della sessione per avere una interazione coerente tra di loro.

3.4 Amazon Web Services

Amazon Web Services (AWS) è una piattaforma pubblica basata su cloud che offre servizi gestiti per la generazione di soluzioni a diversi livelli di astrazione. Per quanto riguarda lo storage ad esempio, si potrebbe richiedere al provider una macchina virtuale e collegare ad essa un volume, implementando così una soluzione di basso livello. Altrimenti è possibile sfruttare servizi come S3 e salvare le proprie informazioni attraverso una API, ottenendo un risultato simile, ma ad alto livello. La scelta dipende sempre dalle necessità dell’utente, tuttavia l’opzione più facile è sicuramente la seconda, perché con una semplice chiamata si possono salvare le proprie informazioni come se si utilizzasse un filesystem remoto, senza preoccuparsi della gestione interna, della durabilità, delle dimensioni dei file ed in generale della grandezza dello storage, che può scalare teoricamente in maniera indefinita.

Le API sono accessibili attraverso protocolli web, coerente con le caratteristiche fornite da NIST per quanto riguarda il cloud computing. Tra queste infatti è presente il concetto di "broad network access", cioè le funzionalità del cloud computing vengono erogate attraverso la rete. Altre proprietà comprendono la possibilità di ottenere i servizi su richiesta da parte dell'utente senza ausili esterni (on-demand self-service), condivisione di un'unica piattaforma da parte degli utenti (resource pooling), risposta rapida alla richiesta di risorse imposte dalla variabilità nel workload (rapid elasticity). L'adozione del cloud computing viene spesso sponsorizzata per il modello dei costi lato utente, basata sul pay-per-use. Questo significa ridurre le spese nel caso di startup che non possiedono una infrastruttura IT, oppure per aziende che vogliono ricostruire o estendere la propria. Inizialmente può essere un vantaggio, ma bisogna sempre monitorare l'utilizzo e l'architettura delle proprie soluzioni in modo che i costi operativi non esplodano. In generale quindi il modello introdotto da Amazon può essere considerato un beneficio, tuttavia si tratta del punto di partenza per esplorare ulteriori vantaggi che il provider introduce.

AWS è una piattaforma innovativa, che annuncia costantemente nuovi prodotti. L'implementazione di queste tecnologie viene poi integrata nell'ecosistema esistente, contribuendo al continuo allargamento della piattaforma. I servizi gestiti forniti dal provider sono pensati per risolvere problemi comuni, in modo tale che il cliente si focalizzi solo sulle soluzioni. Per cui non sarà necessario implementare meccanismi di load balancing o per le notifiche in quanto AWS fornisce queste funzionalità a priori. Il paradigma Infrastructure as Code (IaC) rende la gestione delle risorse automatica: si tratta di un effetto collaterale dell'utilizzo delle API. L'utente deve solo dichiarare le risorse necessarie, non gestire manualmente la loro allocazione. Una volta definite le parti del sistema, sarà compito di AWS invocare i comandi che servono per la loro creazione. La capacità del cloud si può adattare in proporzione al carico di lavoro, definendo un numero di macchine virtuali in base alla periodicità (giorni, settimane, mesi) e a quando vengono raggiunti i picchi. Basta incrementare il numero di macchine virtuali e quando non saranno più necessarie eliminarle. Questo meccanismo avviene molto velocemente, quindi non si tratta solo di un vantaggio di scalabilità, ma anche di velocità di allocazione e deallocazione delle risorse. AWS offre dei servizi che sono affidabili by default: ad esempio S3 è progettato per avere una durabilità dei dati al 99,9%. Inoltre vengono forniti strumenti per rendere la propria infrastruttura affidabile, non limitandosi quindi solo ai servizi,

ma anche a livello di sistema. La possibilità di poter deployare la propria infrastruttura in più regioni rientra in questa categoria. Oltre ad essere un vantaggio in termini di accessibilità, latenza e protezione dei dati, quando una delle repliche possiede un guasto, si può continuare a garantire il servizio utilizzando il deploy di un'altra regione. Inoltre, sempre nell'ottica dell'affidabilità, vengono forniti dei servizi per il monitoraggio e l'alerting dell'infrastruttura, come nel caso di Amazon CloudWatch e AWS X-Ray. Si possono così controllare le metriche principali come latenza, utilizzo delle risorse e throughput. Nel momento in cui si verificano anomalie si possono poi impostare degli avvisi per reagire tempestivamente agli eventi inattesi.

Come si può notare in Figura 3.7, i servizi offerti da AWS si dividono in due categorie: hardware e software, basati sull'infrastruttura sottostante[12]. Poiché in questo lavoro si è sfruttato unicamente il modelli Platform as a Services (PaaS) e Software as a Service (SaaS), il focus della classificazione nell'analisi sarà solo sulla parte software.

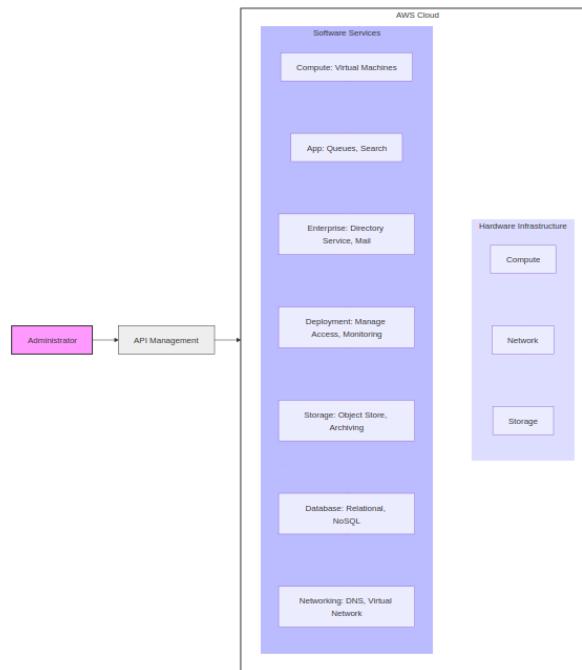


Figura 3.7: Classificazione ad alto livello dei servizi AWS. Accessibili attraverso API, si dividono nelle due macro-aree hardware e software.

Per quanto riguarda la categoria compute, l'obiettivo è quello di fornire rispettivamente servizi per l'esecuzione di applicazioni su macchine virtuali

e per la gestione delle risorse di calcolo, come nel caso di Elastic Compute Cloud (EC2). Per app si intendono quei servizi a diretto supporto delle applicazioni, necessari per la risoluzione di problemi comuni, come l'implementazione delle code di messaggi o delle notifiche. I servizi enterprise sono pensati per la gestione delle risorse aziendali, ad esempio Organizations è un tool per il controllo di più account AWS, in modo tale da poter effettuare raggruppamenti in base alle unità nell'organizzazione ed aggregare policy e operazioni per ciascun gruppo. La categoria deployment, come si può facilmente intuire, fornisce strumenti per la distribuzione del software. Storage è l'insieme dei servizi per la memorizzazione dei dati in modo sicuro, scalabile e di lunga durata. Tra questi rientrano sicuramente gli object store come S3 e i servizi di archiviazione a lungo termine e backup come Glacier. Le ultime due categorie riguardano i database (RDS per i relazionali, DynamoDB per quelli non relazionali) ed il networking come Virtual Private Cloud (VPC) per la creazione di reti isolate nel cloud. In Tabella 3.1, infine, viene mostrata la tassonomia, secondo le classi appena definite, dei servizi usati nell'implementazione del sistema di rilevazione dei DPI, con una breve descrizione del loro funzionamento, reperibile dettagliamente dalla documentazione Amazon[13].

Servizio	Categoria	Descrizione
CloudFormation	Deployment	Tramite il paradigma Infrastructure as Code, permette la dichiarazione di uno stack con le risorse necessarie ed indica le policy di accesso tra i vari servizi.
Lambda	Compute	Funzione utilizzata in architetture event-driven, programmata senza dover gestire l'infrastruttura sottostante (macchine virtuali, sistemi operativi).
IAM Policy	Enterprise	Consente di definire le regole di accesso e autorizzazione tra servizi AWS, rendendo possibili interazioni sicure e controllate.

Kinesis-Video Streams	App	Gestisce in maniera scalabile l'ingestione di flussi multimediali dall'esterno del cloud e li inoltra a servizi downstream, come S3, EC2 o applicazioni di machine learning (e.g. Rekognition).
Kinesis-Data Streams	App	Progettato per gestire flussi di dati non multimediali (JSON, testo), provenienti da fonti come sensori IoT, inoltrati successivamente a servizi big data come Kinesis Data Analytics(KDA) o per lo storage (S3, DynamoDB)
Kinesis-Data Analytics	App	Utilizzato per analisi dei dati in real-time provenienti da flussi generati da Kinesis Data Streams oppure Apache Kafka
Amazon Rekognition	App	Sfrutta modelli di Deep Learning per identificare oggetti, persone, scene all'interno di immagini e video.
AWS IoT Core e Greengrass	App	Controllano in maniera coordinata lo stato dei dispositivi IoT e inoltrano i relativi messaggi di telemetria provenienti dall'edge.

Tabella 3.1: Classificazione dei servizi AWS di interesse.

Fatta eccezione per Amazon Rekognition, tutti questi servizi rientrano nella categoria PaaS, in quanto gran parte della gestione è delegata ad AWS, ma sono presenti delle ulteriori configurazioni perché possano funzionare correttamente. Ad esempio Kinesis Data Analytics richiede di sviluppare applicazioni come Apache Flink e di collegare sorgenti e destinazioni.

AWS possiede un ricco ecosistema per la generazione di soluzioni basate sull'apprendimento automatico, tra cui, oltre a Rekognition, SageMaker e

Bedrock. Ciascuno risponde a esigenze specifiche: SageMaker funge da piattaforma di base per lo sviluppo e l'addestramento di modelli personalizzati, mentre Bedrock offre accesso a diversi foundation models, semplificando l'utilizzo di questa nuova tecnologia. Infine, Rekognition si posiziona come una soluzione specializzata per l'analisi di immagini, video e streaming, dove le aziende possono implementare le relative funzionalità senza dover sviluppare o addestrare modelli. Serve solo invocare la API di interesse. Rekognition è quindi particolarmente utile per le aziende che necessitano di integrazioni rapide e affidabili nell'ambito della visione artificiale all'interno dei loro processi. I casi d'uso spaziano su numerosi domini: ad esempio, può essere utilizzato per estrarre metadati da un testo scritto a mano, oppure per la moderazione dei contenuti nelle piattaforme social.

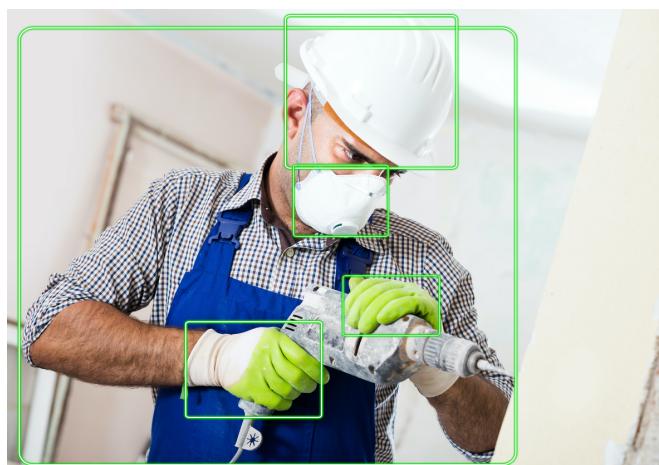


Figura 3.8: Rilevamento tramite Rekognition dei dispositivi di sicurezza individuali.

Nell'ambito di questo scritto, Rekognition viene usato per l'identificazione dei dispositivi di sicurezza e per la valutazione del loro corretto utilizzo. Dalla documentazione Amazon viene mostrato un esempio di questo servizio in azione. La API utilizzata è DetecProtectiveEquipment, che in questo caso è in grado di rilevare casco, maschera e guanti da lavoro. Più nel dettaglio, la risposta di questa richiesta sarà una struttura dati contenente: le persone all'interno dell'immagine, le parti del corpo che indossano i dispositivi di sicurezza (differenziando ad esempio quale mano indossa i guanti), il tipo di dispositivi rilevati e l'associazione tra parti del corpo e dispositivi, in modo tale da verificare quali siano correttamente indossati. La chiamata può essere

configurata per rilevare tutti i DPI più comuni oppure solo un sottoinsieme di essi in base alla necessità. Le API di Rekognition si differenziano in due possibili sottocategorie: storage e non-storage. La distinzione consiste nel fatto che il servizio può salvare o meno le informazioni relative all’analisi dell’immagine o del video. Per questioni di privacy, non viene tracciato alcun individuo e non c’è alcuna correlazione tra gli id restituiti da ciascun processamento. L’operazione è focalizzata solo sull’utilizzo regolare dei DPI e la documentazione è molto chiara in questo punto. Altre chiamate di Rekognition, come quelle basate sul riconoscimento facciale, hanno invece bisogno di salvare queste informazioni altrimenti non potrebbero funzionare.

3.5 Apache Flink

Apache Flink è un framework open source nato per l’elaborazione intensiva dello streaming di dati in real-time. Le sue performance sono dovute a come questa tecnologia è stata pensata. In primo luogo, a differenza delle architetture applicative tradizionali, i software basati su Flink non condividono lo stato esternamente: nei sistemi transazionali, in cui molte applicazioni si basano sulla stessa database, una piccola modifica al modello dei dati può causare molti problemi, se non si opera con la massima accuratezza. Una soluzione è stata quella di introdurre l’architettura a microservizi, dove vengono esposte interfacce REST al posto delle chiamate dirette alla base dati. Nei sistemi di data analysis invece, come quelli basati su data warehouse, sono presenti limitazioni a livello di sincronizzazione e di complessità di trasformazione, in quanto i dati vengono prima estratti da diverse sorgenti per essere centralizzati in un unico punto. Il secondo elemento importante nel design di Apache Flink, strettamente correlato al primo, è lo snapshot dello stato, ovvero in meccanismo per recuperare dai guasti. Questo è fondamentale in quanto Flink è un sistema distribuito e può essere soggetto a failure, come ad esempio errori interni ai nodi, eccezioni o problemi di connettività. La garanzia che lo stato non venga perso, avviene tramite il checkpointing, cioè il salvataggio dei dati in un store remoto. Non si tratta solo di un meccanismo di prevenzione, ma un elemento critico per il funzionamento del sistema. In Apache Flink, senza questa garanzia, nessuna applicazione sarebbe in grado di funzionare in maniera consistente, perché l’elaborazione avviene tramite l’utilizzo di finestre temporali, ed è proprio durante questi intervalli, potenzialmente molto lunghi, che si verificheranno dei guasti.

Nell’elaborazione non è importante solamente il concetto di stato, ma soprattutto quello di stream. E’ difficile nel mondo reale trovare domini in cui tutti i dati vengono generati in un colpo solo. La loro creazione è piuttosto l’effetto di un costante flusso di eventi, ad esempio attraverso l’interazione utente su interfacce web e applicazioni mobili, oppure dalle informazioni provenienti dai dispositivi IoT [14]. Sintetizzando i due concetti, si ottiene il pattern fondamentale su cui è basato Apache Flink: Stateful Stream Processing. L’ultimo elemento del core di Apache Flink è il concetto di event time, ma nel contesto di questo elaborato resta marginale, in quanto non è stato sfruttato nello sviluppo del sistema. Esistono infatti due modalità in cui i dati possono essere processati, di cui il più semplice è il processing time. L’applicazione, seguendo questo paradigma, esegue le elaborazioni in base al clock del sistema piuttosto che basarsi sul timestamp degli eventi. Event time processing è un meccanismo utilizzato nel caso in cui ci sia una necessità di ordinamento tra gli eventi, ma nel prototipo costruito non è fondamentale, in quanto è necessario solo capire se dato un certo numero di risultati provenienti dall’analisi del modello di machine learning, questi globalmente superino una certa soglia, senza alcun vincolo temporale.

Il tipo di processamento in Flink non è indirizzato solo verso gli stream, ma anche al batch, che può essere visto come un caso limite di stream processing. In particolare, i flussi di dati possono essere di due tipologie: limitati o illimitati. Nel secondo caso le informazioni vengono raccolte e successivamente elaborate in blocco. Così Flink, pur essendo un framework nativamente pensato per lo streaming, offre una grossa flessibilità grazie a più modalità di computazione disponibili. In generale nel panorama big data non è lo standard, infatti Apache Spark permette di elaborare grosse quantità di dati, ma con una latenza maggiore, perché non è nativamente progettato per lo streaming. E’ altrettanto vero che sono stati introdotti meccanismi per superare queste limitazioni, come il supporto al micro-batching, ma non riesce comunque a raggiungere le stesse performance per questo tipo di applicazioni.

Nella modellazione dei flussi, viene utilizzato il concetto di dataflow graph, composto da una sorgente, degli operatori ed infine un collettore (sink). Questo tipo di rappresentazione segue una struttura DAG (Direct Acyclic Graph) ed è fondamentale per il framework, in quanto il codice fornito all’applicazione passa per diverse rappresentazioni ed ottimizzazioni intermedie. In particolare, una prima versione logica del grafo viene creata a partire dall’applicativo. Successivamente si passa a quella fisica, ed infine si arriva

all'esecuzione. L'architettura di Apache Flink è composta ad alto livello da tre componenti: il client, quindi il codice scritto dallo sviluppatore, il Job Manager ed infine il Task Manager. Il Job Manager riceve dall'applicazione il grafo di più alto livello e si occupa dell'allocazione delle risorse nell'infrastruttura di deployment. Nella pratica, vengono attivate una serie di Task Manager, in modo tale da dividere il job in diverse elaborazioni parallele. Questo implica sia una suddivisione delle risorse computazionali che un partizionamento dei dati.

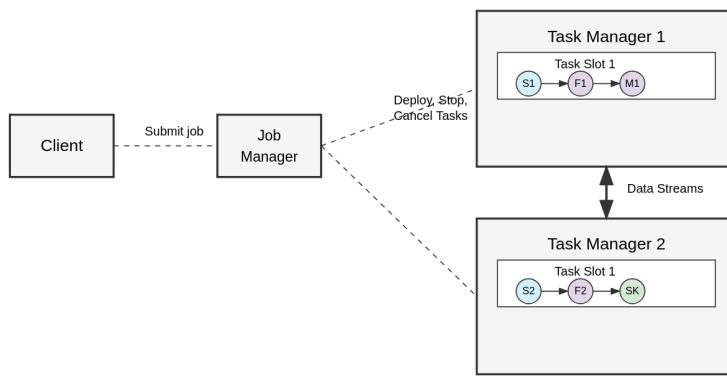


Figura 3.9: Esempio di funzionamento del runtime in Apache Flink.

Come si può notare in Figura 3.9, ogni Task Manager fornisce una serie di task slots, ciascuno dei quali può eseguire una istanza parallela di un grafo. Una volta che il job viene lanciato, il Job Manager rimane responsabile del coordinamento delle attività nel cluster, ad esempio riavvia i Task Manager quando falliscono. Questi ultimi una volta lanciati i jobs, prendono i dati da source differenti, li trasformano e, prima di caricarli sul sink, possono reciprocamente inviarsi le informazioni. Ciò avviene per mezzo di strategie che definiscono come i dati vengono scambiati in un dataflow graph fisico. Possono essere automaticamente invocate dal framework, oppure possono essere esplicitamente dichiarate in fase di scrittura del codice. I pattern principali sono i seguenti:

- **forward:** permette di portare gli elementi dalla sorgente ad un altro punto del grafo. È l'operazione più semplice e meno costosa che si possa applicare.

- **repartition:** si tratta di un'operazione di filtro e raggruppamento. Questo tipo di operazione è più costosa del forward. I dati devono essere serializzati, perché in base a qual è l'operatore di destinazione, potrebbero essere spediti in rete. Un altro operatore infatti potrebbe trovarsi su un nodo diverso da quello di partenza.
- **rebalance:** questa operazione riduce il parallelismo dello stream, ad esempio si potrebbe passare da due flussi ad uno solo. Per l'arbitraggio dei dati che attraversano l'operatore si sfrutta l'algoritmo Round-Robin. Anche in questo caso il costo dell'operazione è rilevante, in quanto di nuovo è necessaria la serializzazione e l'invio dei dati in un altro nodo dell'infrastruttura.

In Figura 3.10 viene mostrato un esempio di dataflow logico, con diverse trasformazioni. Il source può essere di varie tipologie, e nell'ambito di questa tesi viene utilizzato il Kinesis Data Stream connector, per estrarre i record dal servizio gestito Amazon.

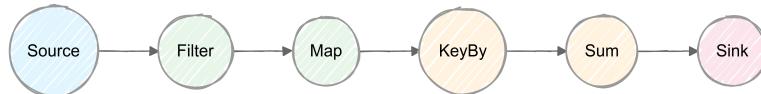


Figura 3.10: Dataflow livello logico.

Questo grafo viene elaborato dal Job Manager, subendo una mappatura nel cluster, come descritto nell'architettura del runtime di Flink in precedenza. Si può notare come alcune trasformazioni sfruttino intrinsecamente i pattern appena descritti.

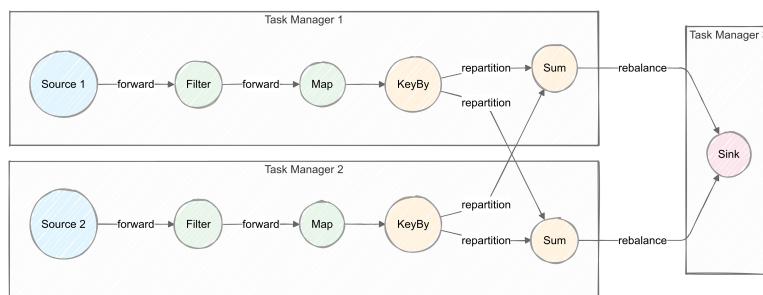


Figura 3.11: Dataflow livello fisico.

Un altro dettaglio, che non viene mostrato nella rappresentazione del dataflow fisico, è il salvataggio dello stato. Ciascun operatore stateful, come le operazioni di aggregazione basate su una chiave di raggruppamento, può infatti persistere localmente informazioni associate a più elementi processati nel tempo.

Il dataflow logico è generato a partire da Datastream API, una delle interfacce possibili fornite dal framework. Apache Flink offre chiamate a diversi livelli di astrazione, in base al tipo di necessità dello sviluppatore. L'insieme di API di più basso livello nello stack è costituito da process functions e Datastream. Le prime possono implementare qualsiasi operazione manipolando direttamente lo stato e i timers. Queste primitive in realtà vengono usate in coppia con Datastram, in quanto le API in Apache Flink non sono esclusive, ma interoperabili. Quindi, dal punto di vista implementativo c'è molta flessibilità, perché nel caso di richieste specifiche nella gestione degli stream, si può passare a chiamate di più basso livello, come nel caso dei timer, che servono per la gestione delle finestre temporali.

Idealmente, si vorrebbe salvare sempre lo stato nelle operazioni stateful, ma questo comporta dei problemi in termini di risorse. Il flusso di dati, come già visto, è indefinito, quindi l'unica soluzione per poter mantenere uno stato di dimensioni accettabili è quello di effettuare le elaborazioni per un numero di eventi limitato. L'utilizzo delle finestre serve esattamente per questo tipo di scenario. Nell'implementazione del sistema sono state usate due tipologie di finestre: quelle basate sulla sul conteggio (count windows) e quelle su timer (time windows).

Capitolo 4

Implementazione del sistema

4.1 Architettura

L’architettura proposta è di tipo event-driven, per cui il processamento dei dati avviene solo quando si verifica una situazione particolare sull’edge, dato il seguente scenario: uno o più operatori entrano all’interno di una certa area definita da un insieme di ancore, su cui sono montati dei sensori e si trovano in prossimità di un macchinario attivo; due telecamere, una superiore ed una frontale monitorano l’area di sicurezza. Se almeno uno degli operatori non possiede i dispositivi di sicurezza, il sistema deve generare un allarme e spegnere il macchinario. Lo stesso tipo di reazione deve verificarsi se almeno uno degli operatori non è abilitato ad operare sulla macchina. Infatti, coloro che possono lavorare indossano un tag attivo che trasmette informazioni ai sensori sulle ancore.

Il design è quello di un’ applicazione cloud nativa e si divide in due sottosistemi tra di loro connessi: da un lato l’edge, composto da una telecamera superiore accoppiata ad una frontale, dei sensori ed un gateway; dall’altro il cloud formato da servizi AWS. Il progetto non è una classica applicazione client-server, ma un insieme di servizi gestiti che globalmente opera near-real time e si serve di dispositivi IoT sia per la rilevazione degli eventi che per l’ingestione dei dati sul cloud. Nel primo caso viene utilizzato MQTT per la telemetria. Nel secondo invece, vengono connesse le telecamere ad un gateway attraverso il protocollo RTSP e successivamente questo si occupa

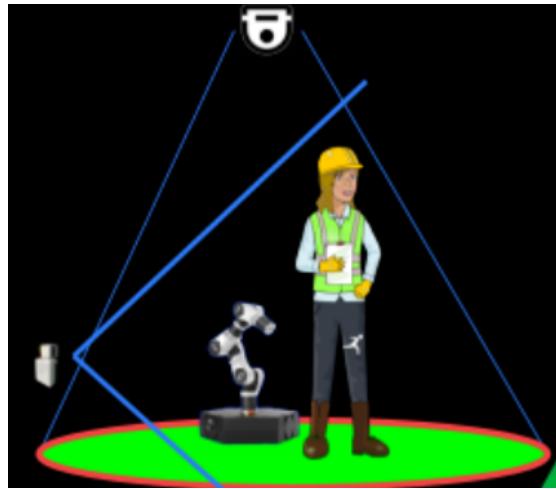


Figura 4.1: Schema dello use case.

dell'ingestione effettiva, sfruttando una libreria Amazon apposita.

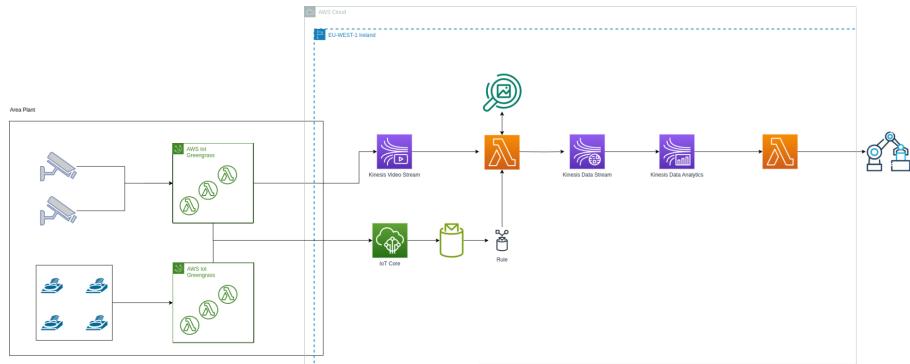


Figura 4.2: Architettura ad alto livello del sistema.

Il gateway posto tra i dispositivi e il cloud non serve solo per garantire la necessaria potenza di calcolo per l'invio dei dati, ma è un componente imprescindibile per l'integrazione con AWS. Al suo interno infatti è installato un software fondamentale per la realizzazione del sistema: IoT Greengrass. Una delle sue funzionalità più importanti è quella di installare sull'edge componenti software caricati dal cloud. Esso consente uno sviluppo del progetto molto più rapido e semplice. Si occupa di problemi quali:

- **sicurezza sull'edge e dei dati in transito:** Greengrass è installato con permessi di root e sfrutta le user permission per l'anti-tampering

del codice dopo il deploy. Per la comunicazione in entrata ed uscita, si serve di certificati relativi alla public key infrastructure (PKI) rilasciati da AWS. Per cui, tramite autenticazione, cifratura e integrità dei dati sia la comunicazione che i deploy sono sicuri di default.

- **orchestrazione del runtime:** i componenti che vengono deployati dal cloud verso l’edge, in qualsiasi forma essi siano (monoliti, servizi o container) possono essere gestiti dall’ambiente, in modo da definire dipendenze tra i vari componenti ed un lifecycle per ciascuno di essi.
- **logging e monitoraggio:** questa integrazione è fondamentale soprattutto per dispositivi che non sono facilmente accessibili da remoto. I log generati dai componenti possono essere automaticamente sincronizzati con servizi cloud AWS come CloudWatch. Inoltre lo stato dei dispositivi è automaticamente tracciato in modo tale da controllare quelli che riscontrano dei problemi e rispondere rapidamente agli errori.
- **scalabilità del deployment:** con Greengrass possono essere dichiarati gruppi di dispositivi su cui deployare dei componenti. Un dispositivo può appartenere anche a più gruppi, così da ricevere diversi deployment contemporaneamente. La definizione di questi insiemi permette quindi un controllo granulare delle istanze per le quali serve fare merging di diversi deployment.

L’obiettivo di questi primi paragrafi è quello di descrivere i componenti hardware e software sull’edge ed i servizi gestiti sul cloud che si occupano dell’ingestion dei dati in maniera sicura, robusta e scalabile, con accenni sulla trasformazione e il preprocessamento dei dati(vedi Figura 4.3). In questa soluzione, Greengrass è installato all’interno di un container Docker che ha come immagine di base la più recente versione di Amazon Machine Image (AMI), pensata per essere integrata con AWS e quindi essere più efficiente e sicura all’interno dell’infrastruttura cloud. Questa macchina virtuale è stata customizzata in modo da supportare sia le dipendenze di Greengrass, che quelle per GStreamer, un tool per l’inoltro dello streaming, che collabora con la libreria apposita per l’invio sul cloud. L’ingestion dei dati video e di quelli provenienti dai sensori è separato ed avviene in due modalità differenti. Una volta che il componente video viene deployato inizia ininterrottamente a streammare su KVS, mentre il publisher che genera l’evento, inviandolo ad un broker su AWS IoT, si attiva solamente alla rilevazione da parte dei sensori di uno o più operatori.

Attraverso Greengrass è possibile generare lo stesso runtime che sul cloud permette l'esecuzione di uno dei componenti più noti in AWS, cioè Lambda, una funzione gestita direttamente dal provider, che elimina la necessità di configurazione di macchine virtuali, sistemi operativi e scalabilità. Il tutto avviene nell'ottica di concentrarsi solo sullo sviluppo della logica per la reazione agli eventi. In questo modo i sensori pubblicheranno i dati su una coda MQTT locale, come il numero di tag rilevati o la macchina cui si fa riferimento, e successivamente la Lambda sull'edge, iscritta al relativo topic, si attiverà alla pubblicazione di queste informazioni. Una volta eseguita una prima elaborazione, la funzione invierà un messaggio su un topic in IoT Core, generando un evento in formato JSON.

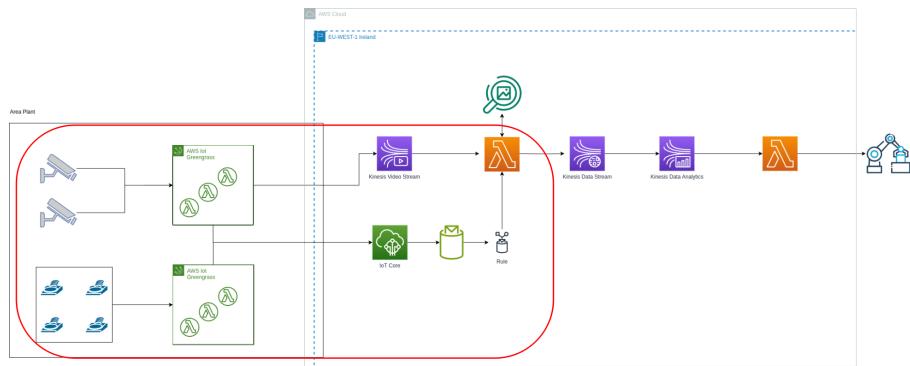


Figura 4.3: Sottosistema di ingestion e preprocessing.

Al messaggio MQTT vengono applicate delle regole, in modo tale da filtrare i dati, successivamente trasmessi ad un'altra Lambda che si trova sul cloud, la quale si occupa del processamento dello streaming video. A valle dell'analisi viene prodotto uno stream di dati verso un'applicazione big data. Nel dettaglio, il flusso iniziale viene separato in due, in quanto la Lambda è progettata per elaborare uno streaming video alla volta ed i flussi streaming da analizzare in parallelo provengono, come già visto, da una telecamera frontale e una superiore. Per cui ogni evento che triggerà la Lambda deve contenere un riferimento allo stream: in questo modo, contemporaneamente, due istanze diverse della stessa Lambda verranno attivate a valle del topic filter.

Per concludere il quadro della descrizione architetturale, bisogna focalizzarsi sui dati in uscita dalla Lambda. Ogni frame analizzato dallo streaming video viene inviato in modalità asincrona sul servizio gestito Kinesis Data

Stream (KDS), il quale permette la trasmissione di grandi flussi di dati in real-time. Queste informazioni possono essere successivamente analizzate in parallelo su dei cluster, dopo aver deployato una opportuna applicazione big data. Si tratta quindi del centro di elaborazione all'interno dell'architettura, in quanto è il punto in cui tutti gli eventi vengono uniti dalle diverse sorgenti IoT, che siano le telecamere o i sensori. In questo progetto è stata sviluppata una applicazione con Apache Flink, come già visto nel capitolo precedente, il framework e motore di processamento distribuito per flussi di dati streaming sia limitati che illimitati. A differenza di altre soluzioni, come Apache Spark, questo framework non solo supporta il batching, ma anche l'analisi di singoli record provenienti da un flusso di dati real-time. Amazon offre un servizio gestito per l'analisi chiamato Kinesis Data Analytics, ma per gli esempi e la ridotta scala dei dati è stato scelto di deployare l'applicazione su una istanza EC2. Il risultato è equivalente e il codice non deve essere modificato per funzionare su un eventuale cluster. Infine, all'estremo del sistema è presente un'ultima Lambda, la cui responsabilità è quella di restituire gli allarmi generati in fase di processamento.

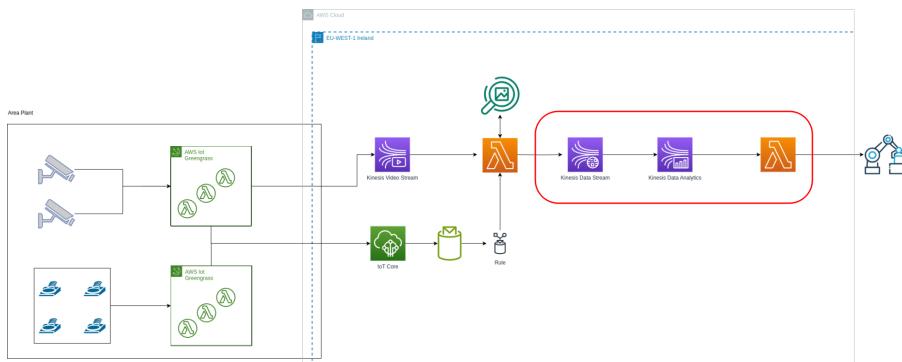


Figura 4.4: Moduli di processamento e generazione degli allarmi.

4.2 Edge Runtime e Ingestion

La soluzione edge, ossia l'insieme dei componenti hardware e software che dall'impianto comunicano con il cloud, segue una topologia a stella. Al centro è posizionato il gateway, il quale si interfaccia con AWS da un lato, e con i dispositivi dall'altro. Più nello specifico, si tratta di una soluzione ibrida, perché uno schema centralizzato dovrebbe essere autosufficiente,

ossia la sua operatività non dipende dallo stato della rete. All'interno del gateway, è installato un Docker container, composto da una serie di strati. Al primo livello è installata l'ultima versione della AMI, formata da un ambiente minimale che si basa sulle distribuzioni Linux Fedora e CentOS. In questo sistema non è possibile installare qualsiasi tipo di libreria, ma solo un numero limitato, fornito da un repository remoto di AWS. Il tutto avviene nell'ottica di una integrazione ottimizzata con il cloud, ma soprattutto per ottenere una sicurezza migliore. Avere un sistema il più possibile ridotto e con un controllo capillare delle installazioni permette di ridurre la superficie di esposizione ad attacchi informatici. Nello strato successivo, vengono instalate delle configurazioni specifiche per Greengrass all'interno dell'ambiente containerizzato. Si sarebbe potuto integrare il runtime direttamente sull'host, ma questo richiede più tempo a causa dei problemi di compatibilità e configurazione. Generando invece un container apposito è possibile replicare questo sistema su tutte le macchine dell'impianto ed in generale distribuirlo a più clienti. Vengono successivamente installate le dipendenze per Greengrass, ad esempio una Java virtual Machine, fornita da Amazon, in modo tale che questo servizio possa essere effettivamente eseguito all'interno del Docker. Contestualmente, in questo strato, una volta che le dipendenze sono pronte, si passa all'installazione effettiva di Greengrass. Lo strato successivo del filesystem è composto dalle librerie per GStreamer, il tool su cui si basa lo stream producer per KVS. Queste dipendenze non sono referenziate dal package manager di AWS, ma indebolendo opportunamente i vincoli di sistema della AMI, è possibile scaricare la giusta versione delle librerie. La customizzazione non è semplice, in quanto gli intervalli di compatibilità tra le librerie del kernel e qualsiasi altra installazione sono molto stringenti. Ad esempio, la regola più forte osservata durante la customizzazione è stata sulla versione di SSL, forzata ad essere quella più recente, ma non supportata dalla maggior parte delle release di GStreamer, sia a livello di dipendenze che di libreria stessa. L'ultimo strato, infine, è composto dall'installazione effettiva di questo tool.

In generale, GStreamer viene usato per costruire una pipeline, ossia un grafo orientato composto da operatori per la gestione di dati multimediali. Il componente software, eseguito sul gateway, internamente sfrutta una libreria che accede a GStreamer. Analogamente, anche le telecamere generano con lo stesso meccanismo i flussi RTSP. Nella configurazione, deve esserci anche un server che utilizza lo stesso protocollo, in modo tale da consentire

l’inoltro dello streaming al gateway. L’ipotesi nello use case è di avere a priori il setup disponibile in fabbrica, con i due flussi RTSP già presenti. Poiché si sta cercando di simulare questo scenario, è stato installato mediamtx, un container di un server multimediale open-source per diversi tipi di streaming, tra cui RTSP. L’obiettivo dovrebbe essere quello di potersi agganciare agli stream ed inoltrarli in maniera sicura verso il cloud. Se le telecamere sono più sofisticate, AWS offre la possibilità di inviare i flussi direttamente su KVS registrando i dispositivi in AWS IoT, ma in generale, come meccanismo standard, è utile convogliare le informazioni sul gateway per diverse motivazioni. In primo luogo, l’utilizzo di un intermediario permette una gestione dello streaming più articolata. Inoltre il gateway possiede risorse di calcolo superiori rispetto a delle telecamere IoT. Questo, ad oggi, è fondamentale nel caso in cui si voglia deploysare dei modelli di machine learning direttamente sull’edge, naturale estensione di questo sistema.

Esistono diversi blocchi per la costruzione della pipeline, ma in questo caso, trattandosi di generazione ed inoltro di streaming, sono stati usati i seguenti elementi: source, sink, filtri e codec. Il source è un elemento che porta i dati dall’esterno verso la pipeline. Ad esempio si può accedere alla telecamera di un computer, file multimediali e soprattutto flussi provenienti dalla rete, basati su protocolli come RTSP ed HTTP. Il sink invece esegue l’operazione contraria: questo operatore può caricare l’output della pipeline in memoria, su dell’hardware esterno come un monitor, oppure inoltrare direttamente il flusso su un endpoint. I filtri e i codec servono per operazioni intermedie, nel primo caso per la modifica dello stream come il framerate o la dimensione del frame, mentre nel secondo per cambiare la codifica delle immagini.

Per quanto riguarda la pubblicazione degli eventi, la Lambda, che si occupa del preprocessing, viene triggerata con un messaggio pubblicato su un topic, come si può notare dall’esempio in Figura 4.5. Rispetto al flusso di dati non strutturato proveniente dalle telecamere, le informazioni dei sensori sono estremamente minimali, in quanto MQTT, essendo un protocollo leggero, è soprattutto pensato per questo genere di messaggi. L’evento è formato da campi per l’identificazione univoca della macchina associata ad una certa area di sicurezza, e allo stesso tempo riferenzia le telecamere che monitorano il luogo di interesse. Si poteva anche generare due messaggi MQTT per identificare gli stream video separatamente, ma l’overhead è poco significativo in termini di spazio. Inoltre, in una configurazione con due

messaggi per lo stesso evento, sarebbe stato necessario replicare le informazioni associate alla macchina e all'area di sicurezza. I topic filter di AWS IoT permettono, alla generazione delle regole, di creare due trigger distinti, verificando che sono presenti due stream diversi nel JSON, quindi la separazione degli eventi avviene direttamente nel cloud. Questo è soprattutto un vantaggio in termini di sincronizzazione, perché i messaggi non rischiano ritardi nella rete, e le due istanze della stessa Lambda verranno attivate ad intervalli praticamente sovrapposti. I messaggi, infatti, si trovano già nella stessa infrastruttura, e l'ordine di grandezza, in termini di latenza, si riduce notevolmente rispetto a quello della rete pubblica.

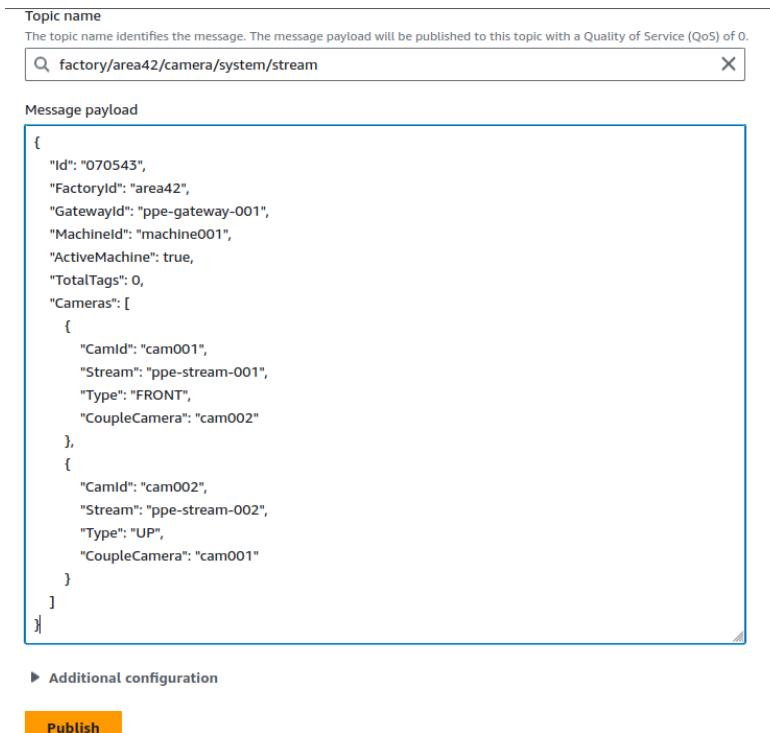


Figura 4.5: Pubblicazione messaggio MQTT sul servizio AWS IoT via console.

4.3 Preprocessing e trasformazione dei dati

La parte di preprocessing, come visto nella descrizione generale, avviene tramite il coordinamento di diversi managed services, il cui fulcro è la funzione Lambda. Essa implementa la trasformazione dello streaming video in

uno flusso di dati in tempo reale, integrando vari servizi AWS in maniera scalabile, efficiente e soprattutto automatica, in base al numero di stream ricevuti. I servizi utilizzati all'interno della Lambda sono Kinesis Video Streams (KVS), Amazon Rekognition, Kinesis Data Streams(KDS) ed optionalmente S3. La funzione è implementata in Python e sfrutta internamente il multithreading per l'ottimizzazione delle prestazioni. Alla base di questo sistema c'è il `lambda_handler`, i.e. l'entrypoint del codice, invocato all'arrivo di un nuovo evento. In termini più specifici, questo tipo di attivazione prende il nome di trigger e viene generato attraverso una topic rule, un meccanismo di filtro fornito da IoT Core. Poiché si tratta del payload di un messaggio MQTT, l'evento viene inviato sotto forma di JSON. Il filtro esegue quindi due operazioni: in primo luogo verifica che il topic sia quello di interesse della Lambda; successivamente può modificare il contenuto del messaggio, scartando alcuni campi, in base a come è stata definita la regola sul servizio. In generale, nella maggior parte dei casi, la comunicazione dei componenti all'interno del cloud è completamente trasparente all'utente e molti servizi non necessitano dell'invocazione diretta delle API, come invece capita normalmente nelle architetture a microservizi. La gestione viene delegata internamente all'infrastruttura AWS. Nelle parti in cui invece l'utente deve scrivere codice, la situazione è diversa: AWS in questo caso fornisce gli SDK in base al linguaggio di programmazione scelto. Nell'implementazione della funzione Lambda si sfrutta la libreria botocore per invocare le API di Rekognition e KDS.

L'algoritmo della Lambda, chiamata Kinesis Video Consumer, inizialmente si occupa di ricevere l'evento, contente il nome del flusso video da processare. Questo dato viene estratto e validato: se mancante, la funzione registra un errore e termina prematuramente. Successivamente viene stabilita una connessione con KVS, per ottenere l'endpoint da cui scaricare il flusso video. A questo punto viene creato un client apposito per accedere ai dati tramite l'API di KVS `get_media`, agganciandosi allo stream da un momento specifico, definito tra le opzioni della chiamata. Poiché l'obiettivo del sistema è quello di elaborare un flusso in tempo reale, questo parametro deve essere impostato sul frame corrente, cioè l'ultimo che è stato caricato nel servizio KVS. Dopo il collegamento all'endpoint e l'inizializzazione del flusso, inizia l'elaborazione del video vera e propria. Lo stream viene processato iterativamente, elaborando i frammenti ricevuti dal KVS. Questa unità contiene al suo interno un certo numero di frames, il cui valore è

configurabile dalla sorgente del flusso multimediale I relativi bytes vengono progressivamente accumulati in un buffer temporaneo, per raggiungere il confine di separazione tra un pezzo di video ed un altro. Una volta che il fragment è stato ricostruito, vengono estratti i frames contenuti al suo interno. Essi sono successivamente codificati in un formato accettabile per la API di Rekognition.

A questo punto ciascun frame viene inviato al servizio per l'analisi delle immagini, in particolare si sfrutta `detect_protective_equipment`, specificando il tipo di DPI da rilevare ed una soglia di confidenza minima. Le chiamate avvengono in parallelo sfruttando il thread pooling, con la relativa struttura dati inizializzata prima dell'esecuzione dell'handler. Le variabili globali definite all'interno della Lambda infatti, permettono di conservare lo stato delle informazioni ad ogni nuova esecuzione della funzione, quindi al verificarsi di un nuovo evento. Quando possibile, è estremamente utile usare questo meccanismo in modo tale da ridurre al massimo il tempo di elaborazione. Inoltre, sempre in termini di efficienza, occorre che i trigger si attivino almeno ogni 15 minuti, in modo tale che non ci sia una deallocazione delle risorse associate alla Lambda. La frequenza degli eventi è imprevedibile, per cui ha senso programmarli al superamento di una certa soglia temporale. È possibile implementare questa logica attraverso i timer impostati con il servizio CloudWatch. Infine, sempre in maniera asincrona, il JSON risultante dall'analisi viene inviato al servizio gestito KDS, invocando una opportuna API.

4.4 Big data analysis

L'applicazione big data è formata da classi per la modellazione dei dati in ingresso, per la rappresentazione dei dati trasformati e per la effettiva elaborazione.

Il primo gruppo di classi, associato al data model, è fondamentale per la rappresentazione dei dati streaming in ingresso all'applicazione, provenienti dall'analisi del video. Il tutto è strutturato in modo da essere modulare e facilmente manutenibile sfruttando il paradigma della programmazione ad oggetti. La composizione delle classi permette la deserializzazione in runtime del JSON proveniente dall'elaborazione di Amazon Rekognition. Esse sono strutturate nel seguente modo:

- **RekoResult:** È la classe principale che rappresenta il risultato di un’analisi video per la rilevazione dei dispositivi di protezione individuale. Contiene la versione del modello machine learning che ha analizzato i frame durante la fase di preprocessing, il numero totale di frame analizzati, un riepilogo dell’equipaggiamento rilevato, informazioni sul dispositivo sorgente ed una lista di persone rilevate.
- **Summary:** Fornisce un riepilogo degli individui rilevati nel video, suddividendo gli identificativi in tre categorie: persone con l’equipaggiamento richiesto, senza l’equipaggiamento richiesto e con equipaggiamento indeterminato. In quest’ultimo caso il modello non è stato in grado di distinguere se un lavoratore indossi correttamente i dispositivi.
- **SourceInfo:** Questa classe contiene informazioni dettagliate sulla fonte del video, relative a telecamera, sito industriale, gateway e macchina di riferimento.
- **Camera:** Modella i dettagli della telecamera utilizzata, come l’identificativo, il tipo, lo stream video, e la telecamera associata, frontale o superiore a seconda del caso.
- **Person:** Rappresenta una persona rilevata, includendo informazioni sulla posizione nel video, un elenco di parti del corpo rilevate, la confidenza del modello nella rilevazione della persona, e un idnetificativo univoco per la persona stessa.
- **BodyPart:** Descrive una parte del corpo di una persona rilevata, associando una confidenza nella rilevazione, e una lista di dispositivi di protezione rilevati su quella parte del corpo.
- **EquipmentDetection:** Indica un dispositivo di protezione rilevato su una parte del corpo, con dettagli sulla posizione, il tipo di equipaggiamento, la confidenza della rilevazione, e se copre adeguatamente la parte del corpo in questione.
- **CoversBodyPart:** Questa classe modella il risultato dell’analisi di copertura di una parte del corpo attraverso il dispositivo di protezione. Include un valore booleano e uno di confidenza su questo tipo di rilevazione.

- **BoundingBox:** Utilizzata sia per le persone che per i dispositivi di protezione, questa classe definisce le coordinate di un'area rettangolare in un'immagine o video.

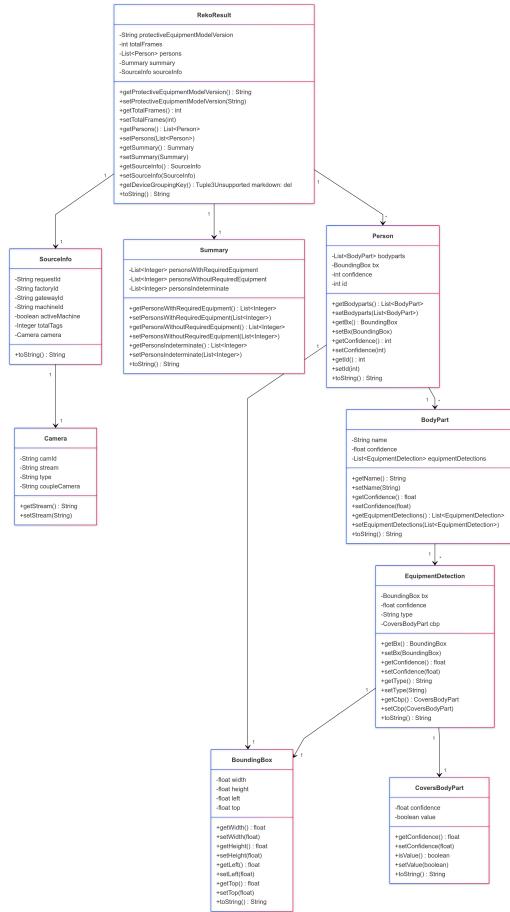


Figura 4.6: Diagramma modello dati.

Le classi di processamento e trasformazione sono associate invece al Job, che come visto nel capitolo precedente, rappresenta il nucleo del sistema di elaborazione. Il dataflow graph logico, generato dall'applicazione, si occupa di processare flussi di dati provenienti da Kinesis Data Stream, i.e. il source. L'analisi in questa prima fase avviene su due flussi differenti, poiché dipende dal tipo di telecamera che ha generato gli stream video. Quella superiore infatti conta il numero di persone nell'area, mentre quella frontale rileva i dispositivi di sicurezza. Una volta connesso alla sorgente, si verifica la lettura dei dati in ingresso, rappresentati in JSON. Successivamente si

applica un filtro, in modo tale da operare solo sulle informazioni relative all’analisi delle telecamere, poiché l’applicazione, in chiave di estensione del sistema, potrebbe ricevere altre tipologie di elementi dallo stesso Kinesis Data Stream. A valle del filtro, un operatore di mapping si occupa della deserializzazione dei dati, il che semplifica l’elaborazione successiva, potendo invocare dei metodi sopra gli oggetti generati, a partire dal data model. Un secondo filtro viene invocato per poter discriminare i flussi delle due telecamere. Si genererebbero altrimenti dei duplicati senza questa distinzione. L’operatore KeyBy si occupa del loro raggruppamento in base al dispositivo, poiché si vuole ottenere in modo coerente l’analisi di un campione di frame. La logica infatti consiste nell’avere una percentuale di immagini, in cui l’accuratezza della rilevazione o del conteggio superi una certa soglia. Ogni telecamera IoT, in un breve intervallo temporale produrrà un certo numero di frame, per cui il raggruppamento è necessario, al fine di distinguere quelli prodotti da ciascuna di esse. Operativamente, l’aggregazione in modalità streaming viene effettuata con le finestre di conteggio. Si poteva utilizzare direttamente la funzione window, fornita dalla API Datastream, ma si è preferito optare per una implementazione di più basso livello. In questo modo, è stato possibile maneggiare direttamente lo stato ed osservare la base della logica di funzionamento per l’implementazione della process function, in coda catena di operazioni seguenti. L’operazione FlatMap permette di poter maneggiare lo stato e le finestre a più basso livello, anche se nativamente non è una trasformazione strettamente stateful.

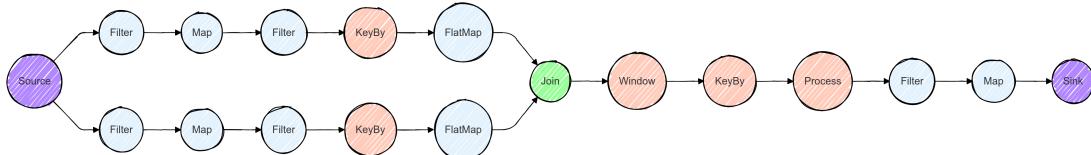


Figura 4.7: Dataflow logico del sistema.

I due datastream vengono poi compattati, creando un singolo flusso, mediante l’applicazione di una finestra temporale. Questa unione permette di correlare i dati provenienti dalle diverse tipologie di telecamere e di sincronizzare i due stream. Per evitare duplicati, un possibile effetto collaterale della finestra scorrevole, i dati vengono ulteriormente filtrati per assicurarsi che ogni richiesta proveniente dall’edge venga processata una sola volta. La fase

finale dell’elaborazione prevede l’applicazione di regole specifiche per generare gli allarmi. Questi vengono prodotti solo quando la macchina monitorata è attiva. Inoltre, le condizioni di sicurezza, definite come 1) una percentuale minima di rilevamento dell’equipaggiamento protettivo e 2) discrepanza nel conteggio delle persone tra sensori e modello di visione artificiale, non sono soddisfatte (da scrivere bene). Gli allarmi così generati sono rappresentati come coppie di valori, e vengono passati a una funzione Lambda per l’elaborazione finale. In particolare, la struttura dati identifica la richiesta originale inviata dall’edge e il timestamp di generazione dell’allarme.

4.5 Deployment

La generazione dell’architettura avviene grazie al servizio CloudFormation, pensato per gestire l’intero ciclo di vita delle risorse AWS. In questo modo è possibile automatizzare tutto un processo storicamente delegato ad un team apposito di IT operations. Attraverso il paradigma IaC, si può, come se fosse un programma, dichiarare tutti i servizi e le risorse necessarie, assieme alle autorizzazioni. Nel prototipo, ad esempio, l’istanza su cui viene caricata l’applicazione big data, non ha accesso a tutti gli object store, ma solo a quello specifico contenente il codice, mediante le policy introdotte in CloudFormation. Nella pratica, in base al livello di astrazione scelto, bisogna scrivere un template YAML o direttamente un programma in un linguaggio a scelta, come Python. A questo punto, dal terminale, si inviano comandi a CloudFormation per mezzo delle sue API. L’interfaccia, dopo aver controllato la sintassi, analizza sia le risorse che i permessi ed internamente invoca tutte le richieste necessarie per la generazione dell’infrastruttura. Se l’operazione va a buon fine, si otterrà un messaggio di successo, altrimenti verrà ritornato errore con un rollback dell’infrastruttura ad uno stato precedente. Formalmente, l’insieme delle risorse create dopo il deploy viene chiamato stack.

All’interno del progetto, sono stati creati diversi moduli di deploy, in modo tale da rendere i componenti riutilizzabili da eventuali altri stack. Core-stack è il template di base, dove vengono definiti i parametri utilizzati nelle altre parti dell’infrastruttura. In questo file viene dichiarato anche il bucket contenente il codice delle applicazioni sul cloud (Lambda, Apache Flink) e quello degli artefatti Greengrass. Streamprocess-stack al suo interno definisce tutti i servizi presenti lato cloud. Components istanzia gli elementi

Greengrass che vengono installati sull'edge. In particolare è uno stack di soli componenti creati dall'utente. Deployments-stack definisce le regole di caricamento dei componenti Greengrass per il gruppo di gateway su cui si vuole effettuare l'installazione. Questa proprietà, come già visto, è molto importante, perché Greengrass e AWS IoT assieme permettono il controllo dell'edge in maniera scalabile. Infatti, oltre a consentire le installazioni da remoto, con pochi comandi, si possono eseguire i deploy su più di una singola macchina.

L'esito della generazione dello stack è visibile sia dal terminale, controllando le risposte dell'API di CloudFormation, oppure direttamente dalla console Amazon, molto più semplice da analizzare. In Figura 4.8 si può notare la presenza di alcuni degli stack definiti in precedenza. In particolare è evidenziata l'infrastruttura relativa ai servizi cloud. Sono visibili diversi campi, i cui più importanti sono gli eventi, gli output e le risorse.

Logical ID	Physical ID	Type	Status	Module
KVSConsumer	KVSConsumerName	AWS::Lambda::Function	CREATE_COMPLETE	-
KVSConsumerRole	KVSConsumerRole	AWS::IAM::Role	CREATE_COMPLETE	-
LambdaLayerVersion	arn:aws:lambda:eu-west-1:21112548922:layer:amazon-kinesis-video-streams-consumer-library-for-python:5	AWS::Lambda::LayerVersion	CREATE_COMPLETE	-
PpeDataStream	ppe-datastream	AWS::Kinesis::Stream	CREATE_COMPLETE	-

Figura 4.8: Stack set dell'infrastruttura di sistema.

I primi sono utili per tenere traccia dello storico di creazione dei servizi, in modo tale che, in caso di errori, si possa individuare la causa del problema. Sempre in questo scenario, verrà eseguito il completo tracciamento del rollback, fino a riportare l'infrastruttura ad uno stack consistente. Errori tipici possono essere delle mancate configurazioni, incoerenza tra le policy, o problemi di dipendenze. Outputs si riferisce agli elementi creati che vengono condivisi tra gli stack. In altre parole questi elementi sono referenziabili attraverso un identificativo logico. Il nome che viene dato alla risorsa di

output è diverso da quello interno allo stack, in quanto essendoci uno scope più esteso, servono delle strategie per l'indificazione univoca dei blocchi di risorse. Tipicamente si identifica l'elemento di output assieme alla regione e al nome dello stack originale. Il campo risorse, infine, indica tutti gli elementi che sono presenti nell'infrastruttura al completamento delle operazioni di creazione, update o rollback. In figura sono presenti alcuni degli elementi descritti nelle sezioni precedenti. KVConsumer è l'identificativo logico nel template per la Lambda, che viene mappato con l'effettiva risorsa presente nel cloud, cioè l'handle fisico. L'interfaccia fornisce un collegamento diretto alla risorsa, in modo tale da esplorare i dettagli del servizio. Alla Lambda viene associata una policy, in questo caso KVConsumerRole: senza questa gestione degli accessi non sarebbe possibile per il servizio operare all'interno dell'infrastruttura. La funzione, grazie al ruolo, può accedere a Kinesis Video Streams, Kinesis Data Streams e può scrivere i log su CloudWatch. Il codice viene caricato sull'object store comune a tutti gli stack, permettendo di separare la logica dell'applicazione dalla configurazione dell'infrastruttura, come succede per tutti i componenti programmabili nel cloud. Un'altra risorsa mostrata è il Kinesis Data Stream, denominato PpeDataStream. Questo flusso è configurato in modalità on-demand, il che significa che la capacità di elaborazione può scalare automaticamente in base alla quantità di dati in ingresso, adattandosi così dinamicamente al workload. Esso, come già visto, svolge un ruolo cruciale nel trasporto dei dati elaborati, fornendo una pipeline affidabile e scalabile per il processamento successivo.

Il secondo stack creato, che viene mostrato nell'immagine, è relativo invece ai deploy sull'edge. Viene definita un'unica risorsa deployment, la quale internamente riferenzia un gruppo di gateway, che devono avere sia Greengrass installato che la registrazione su AWS IoT, tramite dei certificati. Questa risorsa identifica anche l'intero set di componenti che verrà caricato sulle macchine target. All'interno del prototipo il deploy è stato eseguito su un gateway chiamato ppe-gateway-001.

Come si può notare in Figura 4.9, esso deve avere Greengrass installato, infatti nel menù sulla console viene specificato il software con la relativa versione. La thing invece indica che il dispositivo è stato registrato su AWS IoT, il che lo abilita automaticamente a delle comunicazioni sicure. Una caratteristica interessante dei dispositivi di deploy, una volta registrati, è quella di poter scrivere i log di sistema su CloudWatch, come visto anche per la Lambda. In questo caso però, non si tratta solo della possibilità di tracciare i flussi generati dalle operazioni con un tool ottimizzato. Questo

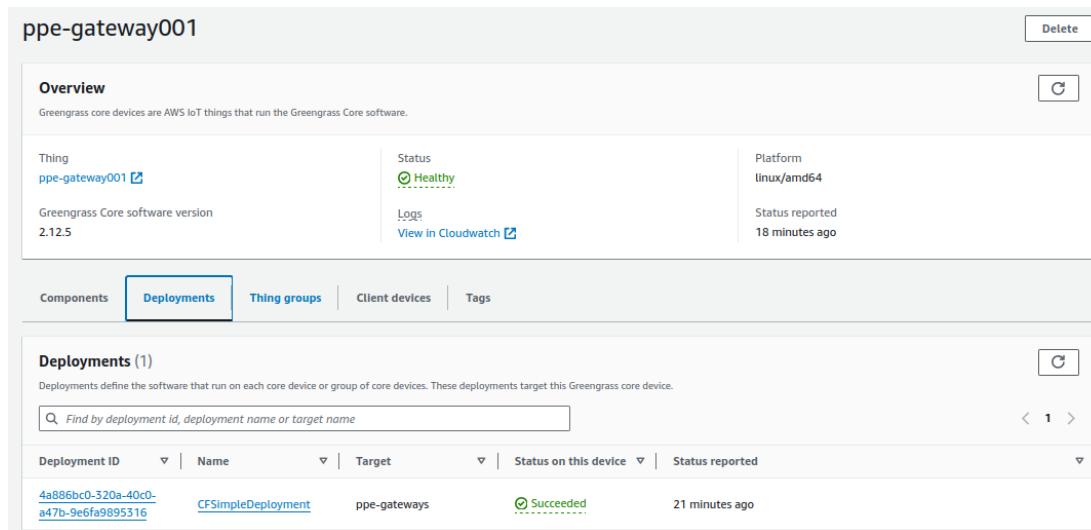


Figura 4.9: Dettagli deployment sul gateway.

meccanismo infatti abilita il controllo remoto delle attività, soprattutto per dispositivi che possono essere difficili da accedere fisicamente. Nell’immagine, infine, vengono mostrate la lista di deployment associati al gateway, il che verifica la possibilità di avere un merge di più installazioni per una stessa macchina o gruppo di calcolatori. In questa trattazione, quindi, si è vista all’opera l’utilità di questo strumento. Senza la necessità di usare manualmente interfacce utente, richiamare direttamente le singole API per la generazione dei servizi o script di vario genere, con una semplice chiamata a CloudFormation si può generare qualsiasi tipo di infrastruttura. Questa può essere replicata facilmente in nuovi account, o in altre regioni, con modifiche minimali. Inoltre, si può notare come sia un tool non limitato solo all’ambiente cloud, ma è integrato anche dove è possibile estendere AWS, come nel caso di Greengrass.

Capitolo 5

Risultati

5.1 Approccio

L’obiettivo del sistema proposto è quello di lavorare near real-time, vincolato sia dai tempi di risposta della API, sia dallo use case presentato nel capitolo precedente. Al momento, si vuole validare la presenza dei DPI indossati da un lavoratore in una zona limitrofa al macchinario. Allo stato attuale, le richieste di analisi dei frame, vengono soddisfatte in tempi superiori al decimo di secondo, il che rende la prevenzione degli incidenti ancora difficile da realizzare. L’approccio utilizzato, come visto nell’implementazione del sistema, è quello di ricevere i dati via streaming, iniziando l’analisi il prima possibile, sfruttando sia il parallelismo nel preprocessing, che la reazione granulare agli eventi nell’applicazione big data. I risultati seguono in maniera coerente questa logica, per cui il focus in questa sezione è più orientato al tempo globale di risposta, ed alla soglia di frame processati correttamente. Questa scelta è motivata da diverse necessità, sotto l’ipotesi che il modello di base abbia delle buone prestazioni, ma che in certe condizioni non performi correttamente. In primo luogo si vuole ottenere robustezza per le detection errate, effetto della generazione di falsi positivi e negativi. Infatti, come visto anche nei lavori correlati, è possibile che un dato oggetto non venga trovato, oppure che la predizione della classe sia errata. In secondo luogo, non avrebbe senso spegnere e riaccendere il macchinario a causa delle fluttuazioni nei rilevamenti, rendendo di fatto l’utilità del sistema poco significativa. Infine, avere un campione di un intervallo di frame permette di compensare potenziali rumori durante la registrazione, come movimenti rapidi, variazioni di angolazione ed occlusioni temporanee.

5.2 Test case e analisi

I test sono stati eseguiti con oggetto solo la rilevazione del casco protettivo, generando un campione che prevede i seguenti parametri:

- **tags**: indica il totale dei tag associati alle persone interne all'area di sicurezza; può non esprimere realmente il numero di lavoratori, in quanto non tutti potrebbero avere il tag aziendale.
- **people**: è il numero di individui effettivamente presenti nell'area di sicurezza.
- **equipment**: definisce se il lavoratore indossa il casco, indipendentemente dall'esito dell'analisi.
- **machine state**: è il valore della macchina all'inizio del test; in questa valutazione è sempre stato impostato come acceso.
- **expected result**: è l'azione attesa a valle dell'analisi, quindi generazione dell'allarme.

I test case sono mostrati in Tabella 5.1 ciascuno di essi è stato lanciato 10 volte. Questo metodo è stato applicato iterativamente in giornate e luoghi differenti, tenendo così conto della variabilità dell'ambiente e della luminosità.

tags	people	eqipment	machine state	Rexpected
0	1	TRUE	TRUE	ALARM
1	1	FALSE	TRUE	ALARM
0	1	FALSE	TRUE	ALARM
2	2	TRUE;FALSE	TRUE	ALARM
1	2	TRUE;FALSE	TRUE	ALARM

Tabella 5.1: Test case.

Il sistema si è comportato correttamente in quasi tutte le prove. Nel primo scenario ha rilevato il dispositivo di sicurezza in tutti i frame per ciascun test. In un solo caso il modello è riuscito a vedere il casco nell'80% dei frame, ma

comunque la detection globalmente è andata a buon fine, perché la soglia di rilevazione complessiva è stata impostata al 70%. Per tutte le restanti prove invece la percentuale di frame ottenuta è stata nulla, in quanto il dispositivo non era indossato almeno da una persona. Per quanto riguarda il numero di persone contate, i risultati sono stati quelli attesi: il modello è sempre stato in grado di effettuare correttamente il conteggio ed il matching è andato a buon fine per tutto il gruppo di frame in ciascuna analisi. Di conseguenza, le regole per l'attivazione dell'allarme si sono sempre verificate.

Per quanto riguarda i tempi di risposta, il sistema rientra nei parametri near real-time, in quanto non ha un framerate superiore a 5fps, considerata la soglia minima per essere classificato come real-time puro. Tipicamente, in questo dominio, viene considerato near real-time un tempo di processamento nell'ordine dei secondi o dei minuti. Il delay tra la prima richiesta a Rekognition e la risposta per l'ultimo frame analizzato è di circa 850ms. L'applicazione big data invece impiega mediamente 550ms per ricevere gli input ed eventualmente inviare l'allarme. Quelle appena viste sono metriche di più basso livello, nell'ottica di quantificare l'impatto di ciascun servizio Amazon sul sistema. La misura più importante, tuttavia, resta la latenza totale dalla generazione dell'evento tramite i sensori, fino al ritorno dell'analisi nell'area di lavoro. La media ottenuta dai campioni generati in fase di test è stata di 2,56 secondi con una deviazione standard di 0,374.

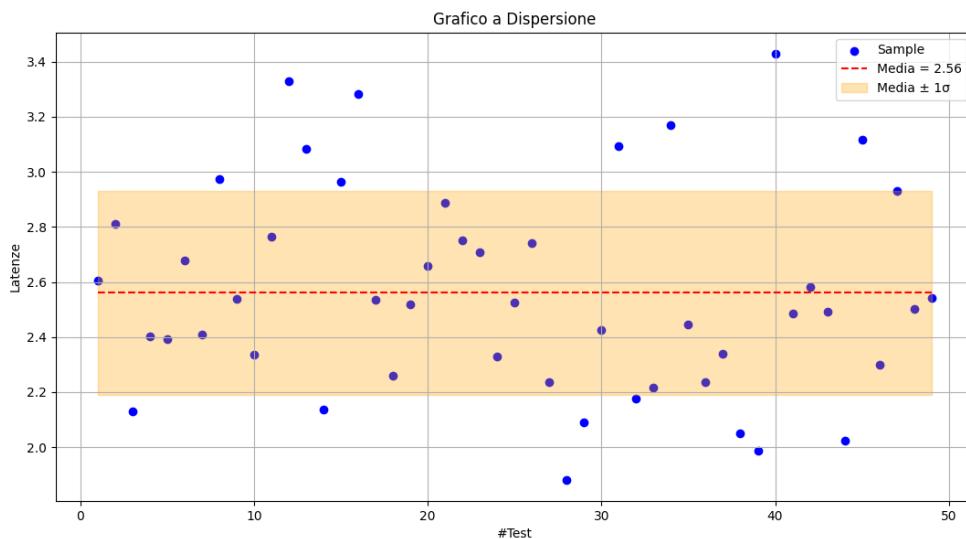


Figura 5.1: Latenza totale del sistema.

5.3 Limitazioni

I test sono stati eseguiti con le immagini acquisite dalle telecamere, ma non sono state svolte in un ambiente industriale, come nel caso dei related works. Inoltre le metriche considerate non sono rapportabili, quindi servirebbero ulteriori indagini in questa direzione. Si tratterebbe comunque di una valutazione imparziale in quanto i modelli di riferimento sono stati installati direttamente nell'area industriale e non si trovano sul cloud. In primo luogo è necessario generare un modello installabile sull'edge, ma Amazon non offre questa possibilità con il servizio Rekognition, pensato per soluzioni che non necessitano allenamenti o installazioni, come visto nella sezione dei servizi AWS. Per potersi allineare ai lavori citati occorre quindi scegliere un'altra opzione che integra edge e cloud contemporaneamente, cioè SageMaker. Sarebbe possibile in questo caso sfruttare l'infrastruttura AWS per semplificare la gestione del ciclo di vita di un modello custom, generato su un dataset ad hoc. Si dovrebbe lo stesso approccio per avere le migliori prestazioni possibili: il modello deve adattarsi ad un ambiente specifico e quindi necessita della generazione di immagini storiche dalla fabbrica e di un processo di etichettatura. Se si dovesse sviluppare l'applicazione near real-time con il deploy sul cloud, le prestazioni sarebbero migliori già in termini di latenza, perché non si sfrutterebbe la API di un servizio gestito, pensato per essere utilizzato in larga scala da un numero definito di utenti.

Un'altra limitazione di questa analisi riguarda l'ambiente di test, che al momento non è automatizzato. Si deve infatti eseguire manualmente degli script e modificare i parametri in base al test case. Per eseguire i test non sono stati utilizzati dei dati provenienti dai sensori, ma viene simulato il JSON che si otterrebbe alla generazione dell'evento per lo scenario specifico. L'analisi dei frame nei test avviene tramite live streaming, ma sarebbe più opportuno definire come source un insieme standard di video storici in modo da rendere i test facilmente ripetibili. L'automatizzazione dell'insieme di test invece sarebbe attuabile tramite degli orchestratori. Nell'ecosistema AWS esiste un servizio per questo tipo di scenario, le Step Functions. Affinché questo workflow venga implementato ogni volta che ci sono modifiche al sistema occorrerebbe integrare il tutto all'interno di una pipeline CI/CD, dove ad ogni commit si invoca l'orchestratore per eseguire tutti i test necessari. Questo meccanismo inoltre non si limiterebbe solo ai test, ma anche al build del codice, al deploy dell'infrastruttura ed infine al monitoraggio. AWS offre un servizio integrato apposito per le pratiche di devops chiamato

CodePipeline. Un drawback relativo alla valutazione del sistema, correlato in parte all'impossibilità di esecuzione in un ambiente industriale, è il fatto di essersi limitato a sole due telecamere. Idealmente, bisognerebbe testare funzionalità e prestazioni su un set maggiore di dispositivi, in modo tale tale da verificare la risposta del sistema in scala. Sarebbe inoltre utile testare in maniera coerente anche l'integrazione con i sensori, che sebbene la generazione simulata degli eventi renda i test attendibili, con buona probabilità non tiene conto di eventuali problematiche che si aggiungono in un caso reale. Ad esempio dovrebbero essere verificati eventuali disconessioni e problemi di sincronizzazione. Infine, un'ultima limitazione di questa analisi riguarda la potenza del gateway: la macchina server utilizzata non ha la potenza di calcolo necessaria per supportare l'interazione con un numero elevato di dispositivi, soprattutto per quel che riguarda l'inoltro dello stream. Infatti durante la valutazione, anche con un ambiente minimale, si è notato un grosso overhead causato soprattutto dai tempi di codifica e decodifica dei flussi video provenienti dai source RTSP.

5.4 Lavori futuri

Eventuali estensioni del sistema dovrebbero in una prima fase rispondere alle limitazioni appena citate, di cui sono già state proposte alcune soluzioni. Successivamente occorre operare in altre aree per il completamento del prototipo, come l'integrazione con i robot. Ciò avviene tramite la generazione di un container apposito, simile a quanto fatto per il gateway, ma direttamente sui dispositivi. La macchina virtuale al suo interno conterrà una immagine basata su un sistema operativo real time per la robotica, ad esempio ROS, sopra il quale dovrà essere presente una installazione Greengrass, per facilitare l'integrazione con il cloud. In questo modo si ottengono tutti i vantaggi del runtime, che nel caso di applicazioni legate alla robotica diventano vitali. La gestione remota della telemetria e gli update per sistemi di questo tipo creano molti problemi ai produttori, sia in termini di affidabilità che di tempo. Generare delle soluzioni apposite infatti richiede un effort notevole alle aziende specializzate in questo campo. L'integrazione con ROS, permette di gestire in maniera molto semplice la comunicazione interna con il robot, in quanto l'invio dei comandi o la comunicazione locale tra varie macchine è coordinata da meccanismi publish/subscribe. L'utilizzo di un container in

quest'ottica offre anche il vantaggio di non interferire con l'ambiente dell'host: un eventuale crash di un'applicazione in questo scenario potrebbe compromettere il funzionamento del robot. Per l'attivazione/disattivazione della macchina quindi bisogna pubblicare l'esito dell'analisi su un topic MQTT di AWS IoT core, al quale i robot interessati saranno iscritti. A questo punto, un componente Greengrass installato sulla macchina si occuperà di processare il messaggio. L'ultima operazione è quella di inoltro del risultato su un topic ROS, che in base alla condizione, eseguirà il comando di accensione o spegnimento.

A livello di processamento big data, Apache Flink offre dei meccanismi più sofisticati nella gestione delle regole. Il sistema attuale, ad esempio, non si adatta alla variabilità delle condizioni nell'ambiente di lavoro. Potrebbero esserci cambiamenti della luminosità, oppure altri effetti che impattano sull'analisi delle immagini, di conseguenza sulle percentuali di frame con rilevazioni corrette. A quel punto, il sistema, dovrebbe essere in grado di aggiornare le soglie tramite comandi remoti, provenienti dai sistemi di monitoraggio, invece di un redeploy con le nuove regole. Il set delle possibili azioni dovrebbe essere salvato in un database che si adatta facilmente a questo tipo di applicazione, come DynamoDB. Per quel che riguarda il tracciamento, allo stato attuale i flussi di log sono molto semplici e coinvolgono solo i preprocessing, ma dovrebbero essere estesi anche all'applicazione big data e al gateway. Il tutto, grazie a CloudWatch, verrebbe gestito in maniera centralizzata e, con riferimento ai device in fabbrica, anche remota.

Infine, l'ultimo sviluppo fondamentale, al di fuori della main business logic, sarebbe quello di generare delle statistiche sul corretto utilizzo dei dispositivi di sicurezza, salvando le informazioni più importanti associate agli eventi di interesse, mostrate su una eventuale interfaccia utente.

Conclusioni

Appendice A

Codice

A.1 Dockerfile

```
1 FROM amazonlinux:latest
2
3
4 # Replace the args to lock to a specific version
5 ARG GREENGRASS_RELEASE_VERSION=2.9.1
6 ARG GREENGRASS_ZIP_FILE=greengrass-`${GREENGRASS_RELEASE_VERSION}.zip
7 ARG GREENGRASS_RELEASE_URI=https://d2s8p88vqu9w66.cloudfront.net/releases/${GREENGRASS_ZIP_FILE}
8 ARG GREENGRASS_ZIP_SHA256=greengrass.zip.sha256
9
10 # Greengrass Version
11 LABEL greengrass-version=${GREENGRASS_RELEASE_VERSION}
12
13 # Set up Greengrass v2 execution parameters
14 # TINI_KILL_PROCESS_GROUP allows forwarding SIGTERM to all
15 # PIDs in the PID group so Greengrass can exit gracefully
16
17 #override with env file during docker generation(defaults
18 #needed in case of missing env file)
19 ENV TINI_KILL_PROCESS_GROUP=1 \
20     GGC_ROOT_PATH=/greengrass/v2 \
21     PROVISION=false \
22     AWS_REGION=us-east-1 \
23     THING_NAME=default_thing_name \
24     THING_GROUP_NAME=default_thing_group_name \
25     TES_ROLE_NAME=default_tes_role_name \
```

```
24     TES_ROLE_ALIAS_NAME=default_tes_role_alias_name \
25     COMPONENT_DEFAULT_USER=default_component_user \
26     DEPLOY_DEV_TOOLS=false \
27     INIT_CONFIG=default_init_config \
28     TRUSTED_PLUGIN=default_trusted_plugin_path \
29     THING_POLICY_NAME=default_thing_policy_name
30 RUN env
31
32 # Entrypoint script to install and run Greengrass
33 COPY "greengrass-entrypoint.sh" /
34 COPY "${GREENGRASS_ZIP_SHA256}" /
35
36 # modify /etc/sudoers and copy configuration for runtime (
37   launch after docker run)
37 COPY "modify-sudoers.sh" /
38 RUN yum update && yum install -y sudo && chmod +x /modify-
39   sudoers.sh && ./modify-sudoers.sh && rm /modify-sudoers.sh
40
41 # Install Greengrass v2 dependencies (python version for
42   components and lambdas = 3.9)
43 RUN yum update -y && yum install -y tar unzip wget procps
44   which shadow-utils && \
45     yum install -y java-11-amazon-corretto-headless && \
46     wget ${GREENGRASS_RELEASE_URI} && sha256sum -c ${
47       GREENGRASS_ZIP_SHA256} && \
48     rm -rf /var/cache/yum && \
49     chmod +x /greengrass-entrypoint.sh && \
50     mkdir -p /opt/greengrassv2 ${GGC_ROOT_PATH} && unzip
51       ${GREENGRASS_ZIP_FILE} -d /opt/greengrassv2 && rm
52       ${GREENGRASS_ZIP_FILE} && rm ${GREENGRASS_ZIP_SHA256}
53
54 # install gstreamer dependencies
55 COPY "gst-plugin-deps.sh" /
56 RUN sudo chmod +x /gst-plugin-deps.sh && ./gst-plugin-deps.sh
57   ${PACKAGES_BASE_URL_UPDATED} ${PACKAGES_BASE_URL_AMLLEVEL}
58   ${ARCH_UPDATED} ${ARCH_AMLLEVEL} && \
59 rm /gst-plugin-deps.sh
60
61 ## install gstreamer: stream raw images and video from the
62   edge, the edge component is based on this software
63 #gstreamer1 (higher fedora level(37) in order to support
64   latest openssl(openssl latest is a system requirement from
65   amazonlinux 2023)
66 COPY "gst-plugin-libs.sh" /
```

```
56 RUN  sudo chmod +x gst-plugin-libs.sh && ./gst-plugin-libs.sh  
      ${PACKAGES_BASE_URL_UPDATED} ${ARCH_UPDATED} && rm /gst-  
      plugin-libs.sh  
57  
58  
59 ENTRYPOINT ["/greengrass-entrypoint.sh"]
```

A.2 Kinesis Video Streams Consumer Lambda

```

1 import os,io, cv2,sys,time,json,boto3,timeit,random,logging
2 from datetime import datetime
3 from concurrent.futures import ThreadPoolExecutor
4 from amazon_kinesis_video_consumer_library.
5     kinesis_video_fragment_processor import
6     KvsFragementProcessor
7 from amazon_kinesis_video_consumer_library.ebmlite import
8     loadSchema
9
10
11 REGION='eu-west-1'
12 #KVS_STREAM01_NAME = 'kvs_camera_stream'    # Stream must be
13             in specified region
14 DATA_STREAM_NAME = "ppe-datastream"
15
16 s3 = boto3.client("s3", region_name=REGION)
17 kds_client=boto3.client('kinesis')
18 kvs_fragment_processor = KvsFragementProcessor()
19 session = boto3.Session(region_name=REGION)
20 kvs_client = session.client("kinesisvideo")
21
22 schema = loadSchema('matroska.xml')
23 reko_client=boto3.client('rekognition', region_name='eu-west
24             -1')
25 min_confidence=70
26
27 #ideally, max_threads = fragmentframes/frames_ratio
28 max_threads=5
29 executor = ThreadPoolExecutor(max_threads)
30
31 def lambda_handler(event, context):
32     futures = []
33
34     try:
35         logger.info('Lambda function started.')
36         logger.debug(f'Event received: {event}')

```

```
36
37     ##### GET IOT EVENT #####
38     print(event)
39     msg_payload = event
40
41     if "Stream" not in msg_payload["Camera"]:
42         logger.error('Missing or invalid Stream in the
43         input payload.')
44         return
45
46     ##### GET STREAM TYPE#####
47     KVS_STREAM01_NAME= msg_payload["Camera"]["Stream"]
48     logger.info(f'Processing stream: {KVS_STREAM01_NAME}')
49
50     ##### GET THE KVS ENDPOINT PROVIDING VIDEO STREAM
51     #####
52     try:
53         get_media_endpoint = kvs_client.
54         get_data_endpoint(
55             APIName="GET_MEDIA",
56             StreamName=KVS_STREAM01_NAME
57             )['DataEndpoint']
58     except ClientError as e:
59         logger.error(f"Failed to get data endpoint
60         for stream {KVS_STREAM01_NAME}: {e}")
61         return
62
63     print(get_media_endpoint)
64
65     ### CONNECT A CLIENT TO THE ENPOINT #####
66     kvs_media_client = session.client('kinesis-video-
67     media', endpoint_url=get_media_endpoint)
68
69     print(kvs_media_client)
70
71     ##### RECEIVE VIDEO FRAGMENT FROM THE KVS STREAM
72     #####
73     try:
74         get_media_response = kvs_media_client.
75         get_media(
76             StreamName=KVS_STREAM01_NAME ,
77             StartSelector={'StartSelectorType': 'NOW',
78             })
79
80
81
```

```

72         except ClientError as e:
73             logger.error(f"Failed to get media from
74 stream {KVS_STREAM01_NAME}: {e}")
75             return
76
77         print(get_media_response)
78
79         ##### OBTAIN FRAMES FROM FRAGMENT #####
80         frames = process(get_media_response,
81 KVS_STREAM01_NAME)
82         frames_nr = len(frames)
83         if frames_nr == 0:
84             logger.warning("No frames extracted from the
85 media fragment.")
86             return
87
88         logger.info(f'{frames_nr} frames extracted from the
89 media fragment.')
90
91         ##### ANYLIZE AND SEND TO KINESIS DATA STREAM
92         #####
93         for idx,fr in enumerate(frames):
94
95             img_jpg = cv2.imencode('.jpg', fr)[1]
96
97             ##### OPTIONAL: SAVE FRAMES TO S3
98             # BUCKET#####
99             #     print(img_jpg)
100            path = "images/" + str(int(time.time()))+_+str(
101 random.randint(1,100))+".jpg"
102            #     print(path)
103            if idx==0 or idx==4:
104                try:
105                    s3.put_object(Body=bytearray(img_jpg),
106 , Bucket="ggtestdeployment", Key=path, ContentEncoding='
107 base64')
108                    logger.info(f'Frame {idx} saved to S3
109 at {path}.')
110
111                except ClientError as e:
112                    logger.error(f"Failed to upload frame
113 {idx} to S3: {e}")
114                    # Decide whether to continue or log
115                    and return
116                    continue

```

```

105
106         ##### SEND FRAMES ASYNCHRONOUSLY TO REKOGNITION
107 #########
108     try:
109         futures.append(executor.submit(push_reko,
110             img_jpg, frames_nr, msg_payload))
111         logger.debug(f'Frame {idx} submitted for
112             Rekognition analysis.')
113     except Exception as e:
114         logger.error(f"Failed to submit frame {idx} for Rekognition analysis: {e}")
115     #futures[idx].result() #don't wait here for
116     #results
117     #response = push_reko(img_jpg) #sequential:
118     #decreases efficiency
119
120
121     ##### PUT EACH ANALYZED FRAMES INSIDE KINESIS DATA
122     STREAM FOR FURTHER PROCESSING #####
123     for idx, future in enumerate(futures):
124         try:
125             result = future.result()
126             print(result)
127             kds_client.put_record(
128                 StreamName=DATA_STREAM_NAME,
129                 Data=json.dumps(result),
130                 PartitionKey="ppe-gateway001-"
131             )
132             logger.info(f'Result for frame {idx} sent
133             to Kinesis Data Stream.')
134         except ClientError as e:
135             logger.error(f"Failed to put record for
136             frame {idx} into Kinesis Data Stream: {e}")
137
138     ### TRACK ANY OTHER POSSIBLE ERROR #####
139     except Exception as e:
140         logger.critical(f'Unhandled exception: {e}', exc_info
141 =True)
142
143
144 def push_reko(img_jpg, frames_nr, msg_payload):
145
146     response = reko_client.detect_protective_equipment(Image
147 ={'Bytes': bytarray(img_jpg)},

```

```
138             SummarizationAttributes={
139                 MinConfidence': int(min_confidence),
140                 RequiredEquipmentTypes': ['HEAD_COVER']})
141             response.pop('ResponseMetadata', 'No Key found')
142             response["TotalFrames"] = frames_nr
143             response['SourceInfo'] = msg_payload
144
145         return response
```

A.3 Apache Flink Stream Process Job

```

1 package SiP; //Safety in Plants
2
3 ...
4
5 public static void main(String[] args) throws Exception {
6     // set up the streaming execution environment
7     final StreamExecutionEnvironment env =
8         StreamExecutionEnvironment.getExecutionEnvironment();
9
10    ObjectMapper mapper = new ObjectMapper();
11    /* if you would like to use runtime configuration
12       properties, uncomment the lines below
13       * DataStream<String> input =
14       createSourceFromApplicationProperties(env);
15       */
16
17    DataStream<String> input =
18        createSourceFromStaticConfig(env);
19
20    /** check if at least PERCENTAGE of frontal images
21     has confidence
22     * above MIN_CONFIDENCE(from rekognition) for head
23     cover being detected and worn
24     **/
```

DataStream<PPEPercentageData> percentages_ppe = input

```
.filter(s -> s.contains("ProtectiveEquipmentModelVersion"))
)
    .map(e -> {
        try {
            return mapper.readValue(e,
RekoResult.class);
        } catch (IOException ioException) {
            System.err.println("Error
deserializing JSON string: " + ioException.getMessage());
            // Skip the problematic data by
returning null, which is filtered out in the next step
            return null;
        }
    })
    .filter(result -> result != null) // Filter
out nulls caused by deserialization failures
    .filter(r -> r.getSourceInfo().camera.type.
equals("FRONT"))
```

```

31         .keyBy(RekoResult::getDeviceGroupingKey)
32         .flatMap(new CountWindowPPEAverage());
33
34     /** check if at least PERCENTAGE of up camera images
35     has
36         * number of tags equal to number of people
37         */
38     DataStream<PeopleCountPercentageData>
39     percentages_people_count = input.filter(s -> s.contains("ProtectiveEquipmentModelVersion"))
40         .map(e -> {
41             try {
42                 return mapper.readValue(e,
43 RekoResult.class);
44             } catch (IOException ioException) {
45                 System.err.println("Error
46     deserializing JSON string: " + ioException.getMessage());
47                 // Skip the problematic data by
48     returning null, which is filtered out in the next step
49                 return null;
50             }
51         })
52         .filter(result -> result != null) // Filter
53     out nulls caused by deserialization failures
54         .filter(r -> r.getSourceInfo().camera.type.
55     equals("UP"))
56         .keyBy(RekoResult::getDeviceGroupingKey)
57         .flatMap(new CountWindowPeopleMatchAverage())
58 ;
59
60     //percentages_ppe.print();
61     //percentages_people_count.print();
62
63     /**
64     * join streams
65     */
66     DataStream<JoinedPPEData> joinedStreams =
67     percentages_ppe.join(percentages_people_count)
68         .where(PPEPercentageData::extractKey)
69         .equalTo(PeopleCountPercentageData::
70     extractKey)
71         .window(SlidingProcessingTimeWindows.of(Time.
72     milliseconds(1000), Time(milliseconds(100)))
73         // .window(TumblingProcessingTimeWindows.of(
74     Time(milliseconds(1500)))

```

```

63         .apply((JoinFunction<PPEPercentageData,
64 PeopleCountPercentageData, JoinedPPEData>)
65             (percentage_ppe, percentage_people_count)
66             -> new JoinedPPEData(
67                 percentage_ppe.getRequestId(),
68                 percentage_ppe.getFactoryId(),
69                 percentage_ppe.getGatewayId(),
70                 percentage_ppe.getCamId(),
71                 percentage_ppe.getCoupleCamera(),
72                 percentage_ppe.getActiveMachine(),
73                 percentage_ppe.getPPEPercentage(),
74                 percentage_people_count.
75                 getPeopleCountPercentage()
76             ),
77             TypeInformation.of(new TypeHint<
78 JoinedPPEData>() {})
79         );
80
81         //joinedStreams.print();
82         /**
83          * sliding window produces duplicates => emit only
84          * one value for the same request
85          */
86         DataStream<JoinedPPEData> result = joinedStreams
87             .keyBy(JoinedPPEData::joinFilteringKey)
88             .process(new DistinctFunction());
89
90         result.print();
91
92         /**
93          * apply rules to generate alerts
94          */
95         DataStream<Tuple2<String, String>> alerts =
96             result.filter("//condition to create alert:
97                         //1 machine active
98                         //2 ppe condition not
99                         satisfied or people and tags number not matching
100                         r -> r.getActiveMachine
101                         () && (r.getPPEPercentage() < PERCENTAGE || r.
102                         getPeopleCountPercentage() < PERCENTAGE))
103                         .map(
104                             r -> new Tuple2<
105                             String, String>(r.getRequestId(), Instant.now().toString())
106                             ).returns(TypeInformation.of(
107                             new TypeHint<Tuple2<String, String>>() {}));

```

```
98     alerts.print();
99
100    alerts.map(t -> {
101        try {
102            invokeFunction(awsLambda, functionName, t
103 .f0, t.f1);
104            consecutiveFailures = 0; // Reset the
failure counter on success
105        } catch (LambdaException e) {
106            consecutiveFailures++;
107            System.err.println("Failed to invoke AWS
Lambda: " + e.getMessage());
108            if (consecutiveFailures >=
MAX_CONSECUTIVE_FAILURES) {
109                System.err.println("Max consecutive
Lambda invocation failures reached. Sending alert to
monitor.");
110            }
111        }
112    }
113    return t;
114 }).returns(TypeInformation.of(new TypeHint<Tuple2
<String, String>>() {}));
115
116 env.execute("Flink Streaming Java API For Safety On
Plants");
117
118 }
119 }
```

Bibliografia

- [1] I. N. per l'Assicurazione contro gli Infortuni sul Lavoro (INAIL), “Rapporto annuale inail 2023,” 2023, accesso: 24 settembre 2024. [Online]. Available: <https://www.inail.it/content/dam/inail-hub-site/documenti/2023/09/infografiche-relazione-annuale-inail-2022.pdf>
- [2] O. Safety and H. A. E. Union, “Il valore della sicurezza e della salute sul lavoro e i costi sociali degli infortuni e delle malattie professionali,” 2019, accesso: 26 settembre 2024. [Online]. Available: https://osha.europa.eu/sites/default/files/Summary_Value_of_OSH_and_societal_costs_injuries_and_diseases_IT.pdf
- [3] G. U. della Repubblica Italiana, “Decreto legislativo 81/2008,” 2008, accesso: 18 ottobre 2024. [Online]. Available: <https://www.gazzettaufficiale.it/eli/id/2008/04/30/008G0104/sg>
- [4] G. U. dell’ Unione Europea, “Regolamento (ue) 2016/42,” 2016, accesso: 22 ottobre 2024. [Online]. Available: <https://eur-lex.europa.eu/legal-content/IT/TXT/PDF/?uri=CELEX:32016R0425>
- [5] D. Hubel and T. Wiesel, “Receptive fields of single neurones in the cat’s striate cortex,” 1959, accesso: 26 ottobre 2024. [Online]. Available: <https://pmc.ncbi.nlm.nih.gov/articles/PMC1363130/pdf/jphysiol01298-0128.pdf>
- [6] J. Manyika, L. Woetzel, and R. Dobbs. (2015) Unlocking the potential of the internet of things. Accesso: 24 ottobre 2024. [Online]. Available: <https://www.mckinsey.com/capabilities/mckinsey-digital/our-insights/the-internet-of-things-the-value-of-digitizing-the-physical-world>
- [7] B. Balakreshnan and Others, “Ppe compliance detection using artificial intelligence in learning factories,” 2020, accesso: 26 Novembre 2024. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S2351978920310556>

- [8] I. Yousif and Others, “Safety 4.0: Harnessing computer vision for advanced industrial protection,” 2024, accesso: 26 Novembre 2024. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S2213846324002451>
- [9] P. Lea, *IoT and Edge Computing for Architects*, second edition ed. Packt Publishing, 2020.
- [10] H. Schulzrinne, “Rtp: A transport protocol for real-time applications,” 2003, accesso: 12 Novembre 2024. [Online]. Available: <https://www.rfc-editor.org/rfc/rfc3550.html>
- [11] ——, “Real time streaming protocol (rtsp),” 1998, accesso: 15 Novembre 2024. [Online]. Available: <https://www.rfc-editor.org/rfc/rfc2326.html>
- [12] A. Witting and M. Witting, *Amazon Web Services in Action*, second edition ed. Manning Publications, 2019.
- [13] A. W. Services. (2024) Amazon documentation. Accesso: 19 Novembre 2024. [Online]. Available: <https://docs.aws.amazon.com/>
- [14] F. Hueske and V. Kalavri, *Stream Processing with Apache Flink*, first edition ed. O'Reilly Media, 2019.