

Distributed cluster for Computer Vision

Bruno Esposito

`bruno.esposito2@studio.unibo.it`

Andrea Ingargiola

`andrea.ingargiola@studio.unibo.it`

23 agosto 2025

Distributed cluster for Computer Vision è un'applicazione che si propone di unire la comodità dell'esecuzione basata su attori in Scala con l'ottimizzazione ottenibile dalla programmazione di algoritmi di visione artificiale in C++. Il sistema consiste in un cluster di telecamere posizionate sulla stessa rete su cui viene fatto eseguire un algoritmo di visione artificiale, che comunica i propri rilevamenti a un numero non definito di client. Nello specifico, si sono implementati una serie di attori che consentono di mettere in comunicazione l'output in tempo reale dei processi C++ con i servizi utili all'analisi e alla visualizzazione dei dati. Per raggiungere il risultato finale è stato fatto utilizzo intensivo di Scala (in particolare delle librerie Akka Actor e Akka Stream), di C++ e di tecnologie accessorie che consentissero l'interoperabilità dei moduli come Docker, Gradle e Github Actions.

Indice

1 Obiettivi del progetto	5
1.1 Scenari di utilizzo	5
1.2 Glossario	5
1.3 Politica di autovalutazione	6
2 Analisi dei requisiti	8
2.1 Semplificazioni del backend	8
2.2 Requisiti di business	8
2.3 Requisiti utente	8
2.4 Requisiti funzionali	8
2.5 Requisiti non funzionali	9
2.6 Requisiti implementativi	9
3 Struttura	10
3.1 Domain Driven Design	10
3.1.1 Domain layer	11
3.1.2 Application layer	11
3.1.3 Presentation layer	11
3.1.4 Storage layer	12
3.1.5 Interface layer	12
3.1.6 Distribution layer	12
3.2 Classi	13
3.2.1 ReachableActor	14
3.2.2 CameraManager	14
3.2.3 Supervisor	17
3.2.4 GenericClient	18
3.2.5 DBCoordinator	20
3.2.6 CLIClient	20
3.2.7 Server	20
3.2.7.1 VertxRouter	20
3.2.7.2 GUIBackEnd	21
3.3 Deployment	22
3.3.1 User Node	22
3.3.2 Utility Node	22
3.3.3 Camera Node	22
4 Dettagli implementativi	24
4.1 Strumenti di Akka utilizzati	24
4.1.1 Receptionist	24
4.1.2 Cluster	24
4.1.3 Stream	24

4.2	Applicazione di Computer Vision	25
4.2.1	Flusso Principale dei Dati	25
4.2.2	Componenti Principali	25
4.2.3	Architettura Concorrente e Gestione Thread-Safe	25
4.2.4	Integrazione con Sistema di Gestione Centralizzato	26
4.3	Applicazione Web: Frontend	26
4.3.1	Componente Principale	26
4.3.2	Sistema di Controllo Telecamere	27
4.3.3	Sistema di Streaming Video	27
4.4	Deployment e Configurazione del Sistema ad Attori Distribuito	30
4.4.1	Entry Point delle Applicazioni	30
4.4.2	Configurazione del Sistema Akka	30
5	DevOps	31
5.1	Build automation	32
5.1.1	Domain Layer	32
5.1.2	Application Layer	34
5.1.3	Presentation Layer	35
5.1.4	Storage Layer	36
5.1.5	Interface Layer	36
5.1.6	Distribution Layer	37
5.1.7	Quality Automation e Process Governance	39
5.2	Containerization	40
5.2.1	Domain Layer	41
5.2.2	Interface Layer: Frontend	43
5.2.3	Distribution Layer	45
5.3	Orchestration	47
5.3.1	Ambiente di Sviluppo: Docker Compose e Automatizzazione Locale	47
5.3.2	Ambiente di Produzione: Docker Swarm, Deployment Automatizzato e Monitoring	49
5.3.2.1	Docker Swarm e Configurazione	49
5.3.2.2	Deployment Automatizzato	50
5.3.2.3	Monitoring e Resilienza	50
5.4	Licenza	51
5.5	Continuous Integration: propagazione dei commit, versionamento, release e documentazione	51
5.5.1	Development Flow: Chain Merge Workflow	52
5.5.1.1	Analisi dei Tradeoff	53
5.5.2	Production Release Pipeline: Release Workflow	54
5.5.2.1	Versionamento Semantico e Trigger Logic	55
5.5.2.2	Operazioni di Build Parallelia	56
5.5.2.3	Generazione Documentazione e Deployment	57
5.5.2.4	Sincronizzazione Cross-Branch	57

6 Istruzioni per il deployment	58
6.1 Deployment in Ambiente di Produzione	58
6.1.1 Deployment Automatizzato	58
6.1.2 Deployment Manuale	59
6.1.3 Architettura di Deployment	59
6.2 Deployment in Ambiente di Sviluppo	59
6.2.1 Deployment Automatizzato per Sviluppo	60
6.2.2 Sincronizzazione Incrementale	60
6.2.3 Gestione del Ciclo di Vita di Sviluppo	61
7 Conclusioni	62
7.1 Risultati Raggiunti e Conclusioni	62
7.2 Sviluppi futuri	62

1 Obiettivi del progetto

L'obiettivo principale di questo elaborato, oltre alla realizzazione dell'infrastruttura Scala che farà da base per progetti successivi più orientati alla visione (e quindi in C++), è quello di approfondire gli argomenti del corso di Software Process Engineering. Per questo motivo si è scelto di utilizzare estensivamente sia strumenti avanzati di version control, come le Github Actions e un linguaggio standardizzato per i commit, che tecnologie di build automation come Gradle, che hanno consentito di configurare correttamente le numerose dipendenze negli svariati linguaggi di cui il progetto è composto.

Al termine dello sviluppo si è ottenuto un buon prototipo delle funzionalità "front-end" del programma indipendenti dall'algoritmo di computer vision utilizzato, che sarà implementato nelle prossime iterazioni del progetto per altri corsi. Per questo progetto infatti l'algoritmo scelto non è altro che una demo di OpenCV riadattata per il riconoscimento dei volti in tempo reale e fatto eseguire su un video per testare che la propagazione dei dati rilevati a tempo di esecuzione avvenisse correttamente nel resto del sistema.

1.1 Scenari di utilizzo

Lo scenario di utilizzo previsto ha un duplice scope: l'elaborato del corso di SPE presentato in questa relazione e l'applicazione finale sviluppata iterativamente al termine degli altri corsi.

- Questo progetto in particolare costituisce un framework ad uso interno per creare applicazioni basate su algoritmi di computer vision che possano funzionare con facilità in ambiente distribuito e con una rapida configurazione: per questo motivo è stata posta enfasi sulla fase di deployment, rendendola compatibile con Docker Swarm, e sull'interoperabilità, concentrandosi sullo sviluppo della parte front-end in linguaggi platform-indipendenti, come Scala per la logica e Javascript per la GUI.
- L'applicativo finale, che sarà sviluppato successivamente, sarà pensato per funzionare in ambiente Business, e le sue funzionalità principali lato backend si occuperanno di analizzare il feed delle telecamere per ricostruire l'ambiente 3D tramite tecniche avanzate di visione artificiale tradizionale: i nodi su cui verranno fatte eseguire disporranno di hardware specializzato (schede video CUDA-compatibili) che sarà messo in comunicazione con il resto dei servizi attraverso l'infrastruttura di base ad attori implementata in questo progetto.

1.2 Glossario

Di seguito l'elenco delle voci che vanno a comporre l'ubiquitous language utilizzato sia per il DDD che per questa relazione:

- Sistema: l'intero applicativo in tutte le sue componenti, partendo dall'acquisizione del feed della telecamera fino alla rappresentazione dei dati rilevati dagli algoritmi di visione artificiale: è formato da una parte di front-end e una parte di back-end;

- Back-end: in questo contesto è la parte di Sistema che si occupa di acquisire il video da una serie di telecamere e di fornire in output i risultati grezzi dell'analisi in tempo reale eseguita attraverso l'algoritmo di visione artificiale scelto;
- Algoritmo: parte del programma in C++ che si occupa di elaborare in tempo reale il video acquisito dalle videocamere per fornire informazioni sui soggetti inquadrati. In questo elaborato le informazioni fornite sono posizione e bounding box (contorni) delle persone inquadrate;
- Programma/Processo C++: con questo termine si indica la parte in C++ del back-end nella sua interezza: questo programma si occupa di acquisire il video dalle videocamere, eseguire l'algoritmo, mandare i risultati al front-end e ritrasmettere sulla rete il feed delle videocamere disegnando sull'inquadratura i rilevamenti eseguiti dall'algoritmo;
- Telecamera/Camera: questo termine verrà usato anche per indicare l'intero hardware remoto che, oltre a occuparsi dell'acquisizione del video tramite le telecamere fisiche, si occuperà sia di eseguire il programma C++ contenente l'algoritmo che la parte di front-end che esegue il broadcast dei dati rilevati in tempo reale;
- Front-end: in questo contesto ci si riferisce alla parte di sistema che si occupa di propagare i dati emessi dal back-end, consentendo a un numero non noto a priori di client di consumarli in modo personalizzato a seconda della propria funzione (GUI, database, interfaccia a riga di comando, ecc...). Il front-end è composto da una parte principale in Scala che fa ampio utilizzo del paradigma ad attori e una parte secondaria in Javascript utile alla consultazione dei dati da parte degli utenti tramite GUI web-based;
- Client: con questo termine si indica un generico consumatore di dati emessi dal back-end. Per quanto la funzione con cui i dati vengono consumati possa cambiare, tutti i client si interfaceranno con il back-end nello stesso modo, garantendo interoperabilità tra i diversi servizi messi a disposizione dal sistema.

1.3 Politica di autovalutazione

La qualità del software prodotto è stata attestata tramite un uso estensivo di unit testing, che ha consentito di ottenere una percentuale di code coverage ritenuta sufficiente: ogni componente software è stato testato singolarmente in modo da raggiungere i requisiti di correttezza, di affidabilità e di robustezza necessari.

Gli obiettivi dell'elaborato, in termini di risultati finali, sono stati considerati raggiunti nel momento in cui:

- Le funzionalità di base descritte in fase di analisi dei requisiti sono state implementate e testate con successo;
- è stata raggiunta la piena automazione nel building dell'applicazione completa, composta da una parte C++, una parte Scala e una parte Docker;

- è stata correttamente concepita e implementata un'architettura basata sul Domain Driven Development.

2 Analisi dei requisiti

2.1 Semplificazioni del backend

Per questo progetto l'algoritmo di computer vision utilizzato è un semplice tracciamento delle persone: per quanto semplice e all'apparenza inutile è stato fondamentale per testare la parametrizzazione in tempo reale di un generico algoritmo di computer vision in C++ attraverso il front-end Scala/Javascript. In questo caso, la parametrizzazione consiste nel regolare l'ampiezza della finestra all'interno dell'inquadratura in cui l'algoritmo effettua il tracciamento.

2.2 Requisiti di business

- L'applicazione fornita deve rappresentare un efficace proof of concept che dimostri il buon funzionamento dell'architettura peer-to-peer progettata per soddisfare i requisiti funzionali del programma;
- Dev'essere fornita un'interfaccia grafica che renda l'idea del funzionamento a regime in tempo reale dell'applicativo finale, che consenta di provarne le funzionalità principali e che dimostri la correttezza dell'architettura utilizzata.

2.3 Requisiti utente

- Visualizzare il feed in tempo reale dell'output dell'algoritmo di visione artificiale;
- Interagire con il sistema per accendere/spegnere l'algoritmo su specifiche videocamere;
- Poder configurare i parametri di avvio dell'algoritmo di una specifica videocamera mentre il sistema è già operativo;
- Poder configurare l'operazione di digestione dei dati da parte di un client mentre il sistema è già operativo.

2.4 Requisiti funzionali

- Ogni telecamera collegata al sistema dev'essere configurabile e raggiungibile in modo indipendente dalle altre;
- Ogni client collegato al sistema deve poter personalizzare la visualizzazione dei dati dei rilevamenti;
- Il sistema deve garantire le sue funzionalità a prescindere dalla presenza o meno dell'interfaccia grafica;
- Il sistema deve funzionare in ambiente totalmente distribuito, in modo fault tolerant e suddividendo, quando possibile, il carico di lavoro.

2.5 Requisiti non funzionali

- Usabilità: facilità nell'utilizzo dell'applicativo per configurare e visualizzare l'output delle videocamere;
- Cross Platform: la parte front-end dell'applicativo dev'essere platform-independent ed eseguibile sui 3 principali sistemi operativi: Linux, Windows, MacOs;
- Scalabilità orizzontale: non devono esserci limiti dipendenti dal software nel numero di camere o client collegabili al sistema;
- L'applicazione deve sfruttare un'architettura decentralizzata peer-to-peer in cui l'arresto di un singolo componente non compromette il funzionamento generale;
- Dev'essere possibile mantenere la persistenza dei dati attraverso un database senza che questo diventi un collo di bottiglia per il funzionamento in tempo reale;
- Dev'essere possibile avviare più client su una stessa macchina senza creare conflitti.

2.6 Requisiti implementativi

- La parte frontend dell'applicativo (dall'intercettazione dell'output degli algoritmi di CV in poi) verrà sviluppata in Scala 3;
- L'interfaccia grafica sarà web-based e sarà sviluppata in Javascript;
- Gli algoritmi di CV saranno sviluppati in C++;
- Verrà usato il framework ad attori Akka Cluster(?) per la gestione della logica distribuita e di scambio dei messaggi;
- Verrà usata la libreria Akka Stream per la gestione della propagazione dell'output in tempo reale dalle camere ai client;
- Per il testing verranno usate le librerie AkkaTestKit e JUnit(?)
- Il database sarà di tipo non-relazionale e verrà implementato tramite MongoDB(?) (e relativo driver Scala).
- Il deployment avverrà attraverso Docker Swarm;
- Gradle, Git e Docker verranno usati per la gestione delle dipendenze;
- Le Github Actions verranno usate per garantire la retrocompatibilità dei vari moduli in caso di modifiche.

3 Struttura

3.1 Domain Driven Design

Lo spazio del dominio consiste nell'identificazione delle componenti che vanno a comporre il framework a uso interno sviluppato in questo elaborato, che è stato sviluppato seguendo un'architettura esagonale. Sono stati identificati due contesti:

- il contesto Utente, che rappresenta tutti i componenti utili alla comunicazione tra utenti finali e l'insieme di telecamere sul cui feed viene eseguito l'algoritmo di visione artificiale;
- il contesto Sistema, che racchiude tutti i componenti automatizzati che non richiedono interazioni da parte degli utenti per funzionare.

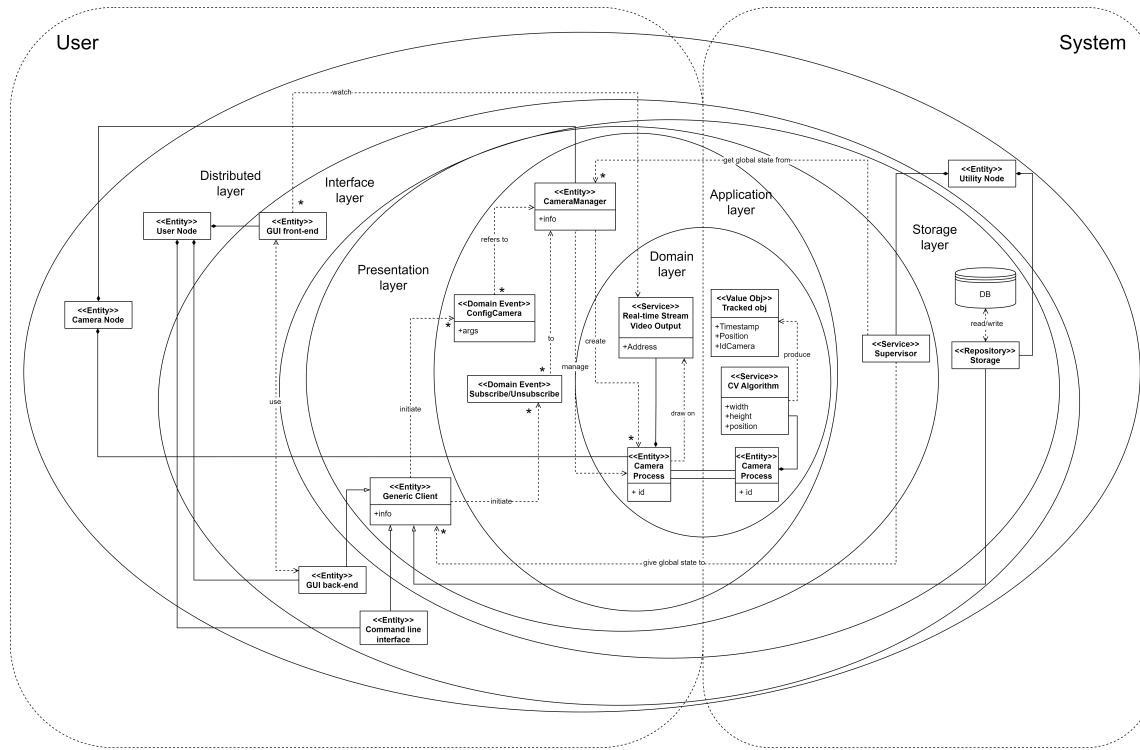


Figura 1: Context map del sistema.

Procedendo dall'interno verso l'esterno, gli elementi di DDD identificati sono:

3.1.1 Domain layer

- **Camera Process** è l'entità che rappresenta il processo C++ che si interfaccia direttamente al feed della telecamera ed esegue l'algoritmo di computer vision. Rappresenta il componente software di più basso livello presente nel sistema, e gestisce sia la rappresentazione grafica dei risultati dell'algoritmo sul feed trasmesso in streaming che il primo passaggio della propagazione dei risultati in tempo reale alla parte front-end dell'applicativo;
- **Real-time Stream Video Output** è l'entità che rappresenta lo streaming del feed della videocamera manipolato dall'algoritmo di visione artificiale (e quindi ad esempio i riquadri colorati intorno alle persone tracciate);
- **CV Algorithm** e **Tracked obj** rappresentano rispettivamente lo specifico algoritmo di visione in esecuzione nel processo C++ e i valori di output che produce, che nel caso dell'algoritmo usato per questo elaborato consiste nella singola persona rilevata all'interno di un singolo frame.

3.1.2 Application layer

- **CameraManager** è il componente Scala che fa da tramite tra il processo C++ e il resto del front-end: si occupa di configurare, lanciare e/o arrestare il processo C++, gli recapita l'eventuale input in formato stringa da parte dei client e gestisce il meccanismo di broadcasting degli output numerico-testuali in tempo reale;
- **ConfigCamera** è il domain event che si verifica nel momento in cui un client qualsiasi manda una nuova configurazione del processo C++ al CameraManager che se ne occupa. Alla ricezione del messaggio di configurazione il CameraManager riavvia il processo C++, parametrizzandolo con i nuovi argomenti in ingresso a riga di comando. Nel caso di questo elaborato i parametri configurabili dell'algoritmo di visione artificiale consistono nell'ampiezza e nella posizione della sottosezione dell'inquadratura al cui interno vengono tracciate le persone;
- **Subscribe** e **Unsubscribe** sono i domain event che rappresentano la sottoscrizione da parte dei client allo stream in tempo reale dei risultati dell'algoritmo di visione artificiale.

3.1.3 Presentation layer

- **Generic Client** è l'entità che implementa il comportamento dell'attore da cui tutti i client erediteranno, in modo che ogni client possa interagire correttamente con gli altri componenti del sistema senza conflitti. Questa classe astratta rappresenta il contratto pubblico a cui le sue sottoclassi dovranno sottostare, implementando

solo le operazioni che le specializzeranno (l'attore Storing per l'interfacciamento col database, il back-end della GUI e l'interfaccia a riga di comando);

- **Supervisor** è il servizio che si occupa di fornire ai client remoti la visione d'insieme del sistema, in modo che tutti i client siano sempre aggiornati su quali camere è possibile sintonizzarsi e quali no.

3.1.4 Storage layer

- **Storage** è l'attore che si occupa di intercettare i dati in tempo reale di tutte le videocamere e registrarli sul database.

3.1.5 Interface layer

- **GUI back-end** è l'attore Scala che si occupa di mettere a disposizione dell'interfaccia grafica in Javascript una API per l'interazione con le videocamere e la visualizzazione dei dati;
- **GUI front-end** è l'entità che rappresenta una singola istanza della GUI Javascript;
- **Command line interface** implementa i metodi astratti del Generic Service per fornire tutte le funzionalità del client GUI tramite un parser a riga di comando.

3.1.6 Distribution layer

- **User Node** è l'attore Scala che si occupa di mettere a disposizione dell'interfaccia grafica in Javascript una API per l'interazione con le videocamere e la visualizzazione dei dati;
- **Camera Node** è l'entità che rappresenta una singola istanza della GUI Javascript;
- **Utility Node** implementa i metodi astratti del Generic Service per fornire tutte le funzionalità del client GUI tramite riga di comando.

3.2 Classi

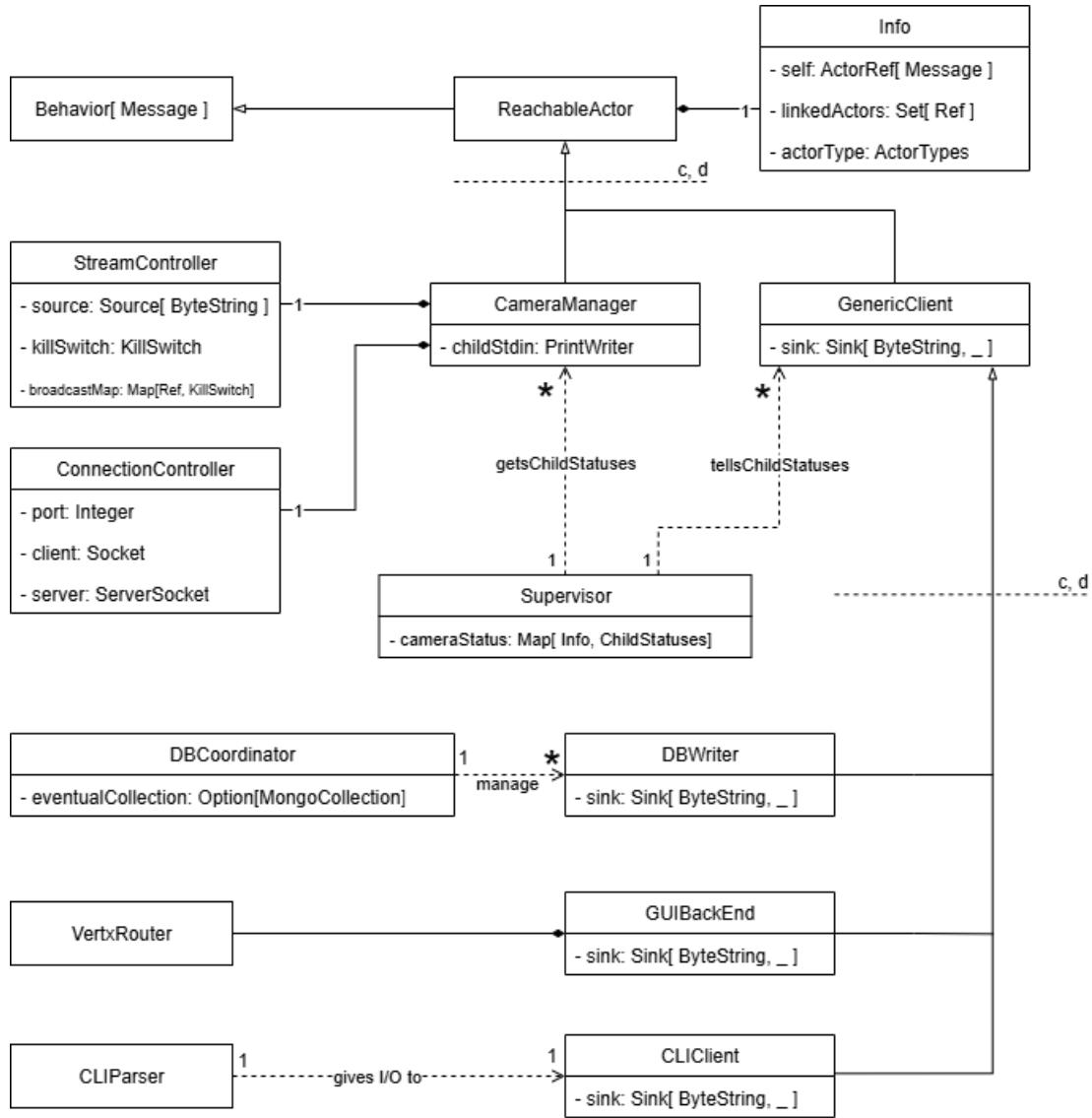


Figura 2: Diagramma UML delle classi della parte Scala del front-end.

Tutti gli attori del sistema sono stati progettati e implementati secondo la filosofia funzionale stateless: possiedono solo un'operazione pubblica, la `create()`, che istanzia un **Behavior** (l'interfaccia Akka che si occupa di specificare il comportamento dell'attore a seconda del messaggio che riceve) ed esegue le operazioni che vanno eseguite solo alla creazione dell'attore (come l'iscrizione all'Akka Receptionist). Al termine della gestione di ogni tipo di messaggio ricevuto, il **Behavior** istanziato richiama ricorsivamente l'o-

perazione privata behavior(), corrispondente al costrutto teorico "become", passando a quest'ultima i nuovi valori dei campi delle varie classi, le cui istanze di fatto saranno composte da un set di campi immutabili. Per fare ciò sono state usate le operazioni apply() dei companion object delle varie classi. Gli attori che ereditano da altri attori aggiungono il proprio Behavior a quello dell'attore superclasse, motivo per il quale è stato fondamentale avere un GenericClient che implementasse tutti i protocolli di comunicazione con il Supervisor e i CameraManager.

3.2.1 ReachableActor

ReachableActor è l'attore che implementa il protocollo di raggiungibilità, ovvero la risposta al messaggio di tipo Ping con un messaggio di tipo Pong contenente le informazioni di quello specifico attore, implementate nella classe Info. Questa classe comprende l'ActorRef dell'attore, un set di eventuali attori collegati (non utilizzato in questo elaborato ma predisposto per il futuro) e il tipo di attore scelto dall'enum ActorType.

3.2.2 CameraManager

Il CameraManager è l'attore Scala che gestisce il processo C++ su cui viene fatto eseguire l'algoritmo di visione artificiale. Possiede tre campi principali immutabili, i cui valori vengono aggiornati in modo funzionale attraverso chiamate ricorsive:

- **ChildStdin** è lo stream di dati in ingresso del processo C++. Grazie alla classe ProcessIO di Scala è possibile agganciare un oggetto di tipo PrintWriter all'esecuzione della riga di comando che lancia il processo da console, in modo che sia possibile simulare l'input da tastiera programmaticamente attraverso le println();
- **ConnectionController** si occupa di creare il lato server della socket tcp utilizzata per intercettare l'output del processo C++. Durante ogni inizializzazione del processo il server verrà resettato e verrà eseguita una nuova accept(), attendendo che il processo C++ si colleghi alla socket come client. Lo stream di input del lato client della socket viene usato come base di partenza per la creazione dello stream da diffondere ai vari Client del sistema;
- **StreamController** è l'oggetto che si occupa di implementare la propagazione vera e propria dei dati in uscita dal processo C++. Per farlo utilizza la libreria Akka Stream per creare una sorgente di dati reattiva a partire dallo stream del lato client della socket di comunicazione tra CameraManager e il rispettivo processo. Questa sorgente, chiamata Source, viene collegata sia a una BroadcastHub, che ne consente la materializzazione multipla, che a un KillSwitch, che ne consente la corretta chiusura. Per collegare un client allo stream è necessario che questo gli invii una SinkRef, ovvero una reference al suo consumatore di dati locale (il Sink). Grazie al BroadcastHub è possibile collegare Source e SinkRef (materializzazione) in modo che i dati emessi dalla Source (e quindi dal processo C++) vengano recapitati anche al client che ospita il Sink, permettendo a quest'ultimo di eseguire

la propria funzione di consumo (ad esempio stampando i dati a schermo o salvandoli nel database).

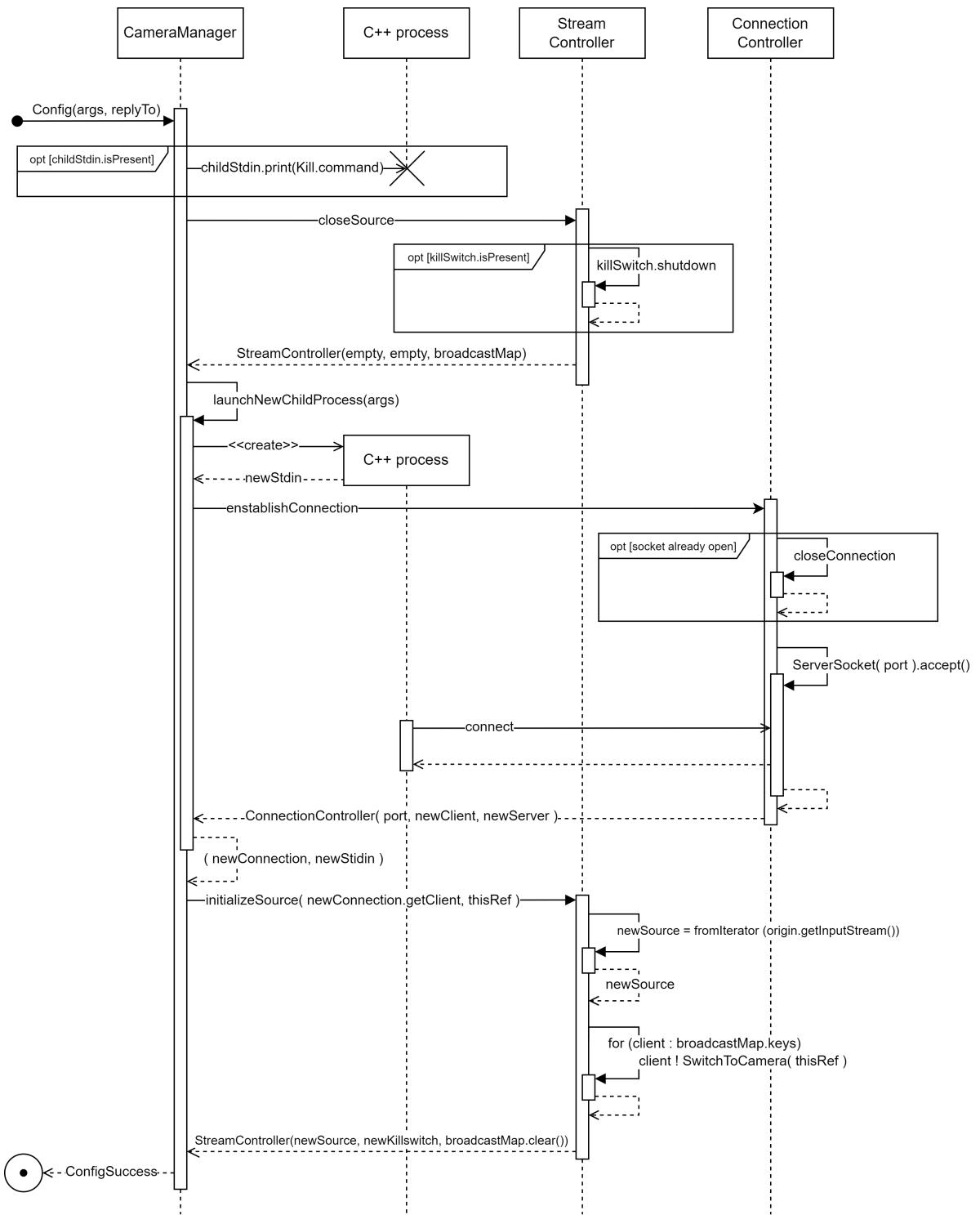


Figura 3: Diagramma UML di sequenza della configurazione del processo C++.

Oltre a disporre del protocollo di Ping (essendo sottoclasse di ReachableActor), implementa le operazioni per i seguenti messaggi:

- **ConfigMsg**: questo messaggio, mandato da un qualsiasi client, parametrizza una nuova istanza del processo C++. Al momento della ricezione il processo viene riavviato con i nuovi parametri (nel caso di questo elaborato, con una nuova sottosezione di inquadratura in cui tracciare le persone): il ConnectionController si occupa di chiudere correttamente la precedente socket usata per la comunicazione tra l'attore e il processo e ricrearla alla stessa porta, mentre lo StreamController usa la sua broadcastMap per comunicare a tutti i client sintonizzati su quella camera la necessità di ristabilire la connessione con lo stream di dati in tempo reale;
- **Subscribe** è il tipo di messaggio con cui i client si sintonizzano con lo stream di risultati fornendo la propria SinkRef: i risultati dell'algoritmo, comunicati dal processo C++ tramite socket, vengono propagati ai client attraverso la libreria Akka Stream, che garantisce prestazioni migliori rispetto ai semplici messaggi del paradigma ad attori utilizzando la programmazione reattiva. Lo stream dei dati in ingresso della socket viene usato come punto di partenza per la creazione di una Source, ovvero un emettitore di dati reattivo a cui vengono allacciati i Sink, cioè i consumatori, inviati dai client tramite il messaggio di Subscribe che ne contiene la SinkRef; Ogni Sink può implementare funzioni diverse, e viene eseguito localmente sui rispettivi client.
- **Unsubscribe** è il messaggio che il client manda al CameraManager per essere cancellato dalla broadcastList del suo StreamController e smettere di ricevere l'output in tempo reale;
- **Input** è il messaggio utilizzato per simulare l'input da tastiera nel processo C++. Nel contesto di questo elaborato serve principalmente per dire al processo di arrendersi attraverso una stringa specifica registrata nell'enum StandardProcessCommands;
- Alla ricezione del messaggio **GetChildStatus** il CameraManager invia al mittente lo stato del proprio processo C++, in modo che sappia se i messaggi di Subscribe e di Input hanno la possibilità di andare a buon fine.

3.2.3 Supervisor

Il Supervisor, tramite l'Akka Receptionist, riceve la lista di tutti i CameraManager, le rispettive info (compresa l'ActorRef) e lo stato dei vari processi C++ per comunicarli a tutti i client del sistema. Tutti i CameraManager vengono raggiunti attraverso la chiave di servizio "inputs", a cui si registrano durante la creazione, mentre i client attraverso la chiave "outputs". Il Supervisor viene creato come Cluster Singleton, in modo che Akka garantisca automaticamente che ne esista sempre una e una sola istanza attiva contemporaneamente in tutte le eventuali partizioni del sistema in caso di guasti di rete.

Questo attore utilizza un MessageAdapter per tradurre i messaggi del Receptionist in sottoclassi di Message, cioè il trait usato come base per tutti i messaggi del sistema.

3.2.4 GenericClient

Questa classe astratta implementa il comportamento di base di tutti gli attori con il ruolo di client nel sistema. Permette di utilizzare il protocollo di sottoscrizione all'output in diretta dei CameraManager e viene costantemente aggiornata sullo stato (acceso/-spento) di tutti i processi C++ attivi. Questa classe permette la personalizzazione delle operazioni svolte alla ricezione dei vari messaggi (oltre al comportamento di default necessario per far funzionare il client) grazie al metodo astratto void onMessage, che viene richiamato per primo ogni volta che viene gestito un messaggio (tranne ConfigureClientSink) passandogli il messaggio appena ricevuto. In questo modo ogni implementazione concreta potrà personalizzare il comportamento di dominio specifico tramite side-effects controllati senza reimplementare il comportamento di base. Il CameraManager a cui il client è sintonizzato è salvato nel campo linkedActors delle sue Info. I messaggi che gestisce sono:

- Esistono due possibili mittenti del messaggio **SwitchToCamera**: l'utente e un CameraManager. Quando il messaggio è mandato dall'utente significa che probabilmente si vuole cambiare CameraManager di cui osservare lo stream. In questo caso la reference contenuta nel messaggio non corrisponderà a quella contenuta nei linkedActors. L'GenericClient allora invierà sia un messaggio di Unsubscribe alla camera corrente che uno di Subscribe a quella nuova. Se invece il messaggio è stato mandato da un CameraManager significa che il processo C++ corrispondente è stato riconfigurato, riavviando quindi la Source su cui questo client era sintonizzato. Il mittente del messaggio non è specificato in quanto non significativo: l'unico controllo che viene fatto per stabilire in quale dei due casi ci si trovi è quello sulla corrispondenza tra la reference della camera a cui ci si vuole sintonizzare e quella a cui il client è attualmente sintonizzato.
- **ConfigureClientSink** è il messaggio generalmente inviato da un utente per stabilire quale sia la funzione di consumo dei dati in arrivo dalle Source dei CameraManager. Gli GenericClient sono inizializzati con una funzione identità, ed è possibile inviarli questo tipo di messaggio sia prima che dopo la sottoscrizione a un CameraManager. La funzione rimarrà configurata anche in caso di cambio di camera sottoscritta.
- **SubscribeServiceSuccess** è il messaggio di risposta a una sottoscrizione avvenuta con successo: è solo alla ricezione di questo messaggio che l'GenericClient aggiornerà il proprio campo info.linkedActors per ospitare la reference del CameraManager a cui è iscritto.
- **InputServiceSuccess**, **SubscribeServiceFailure** e **ConfigServiceSuccess** sono messaggi di servizio di cui è predisposta la gestione da parte delle sottoclassi di GenericClient, senza che nella classe astratta sia fatto alcunché.

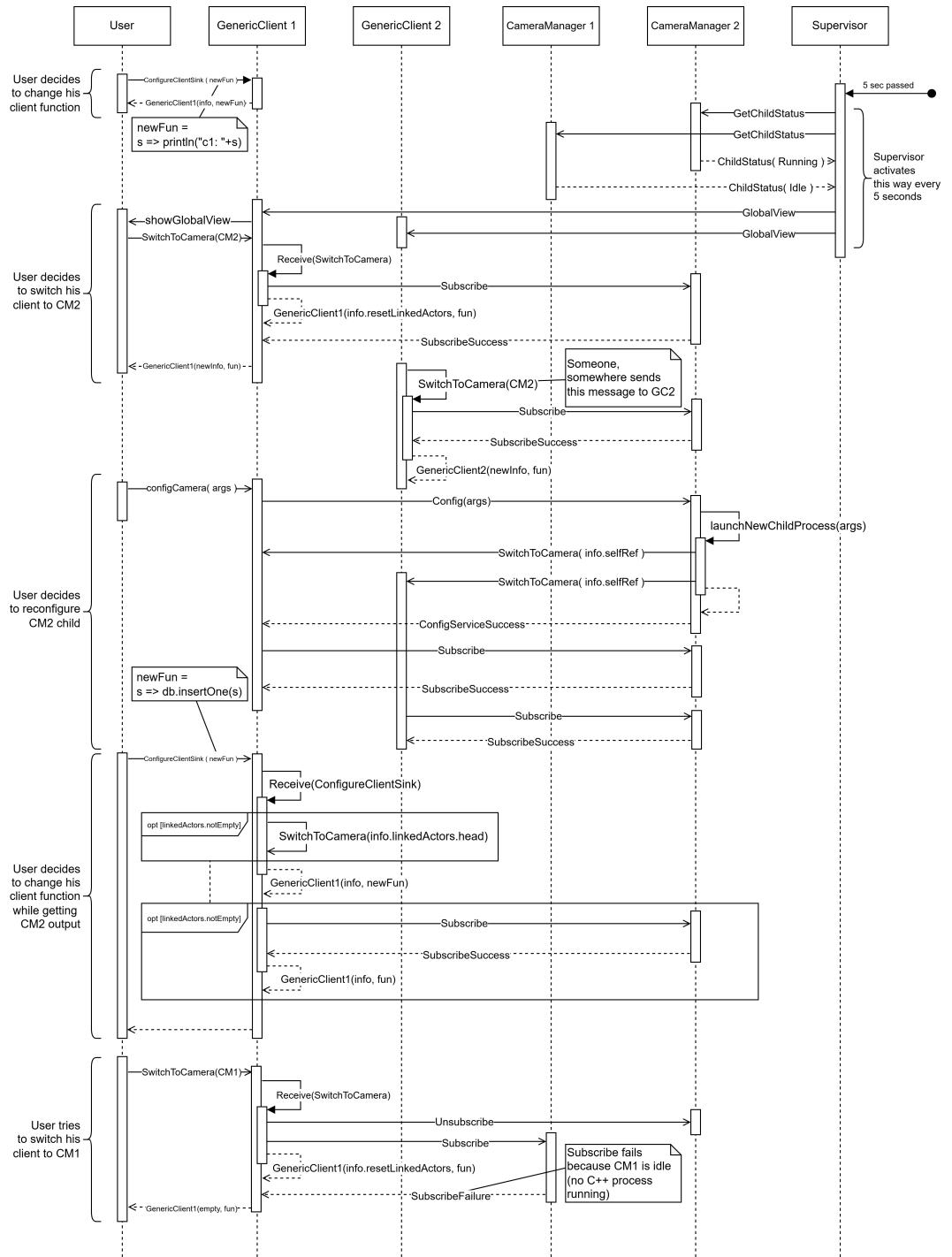


Figura 4: Pseudo diagramma UML di sequenza degli scenari di utilizzo del client.

3.2.5 DBCoordinator

Il DBCoordinator è l'attore principale del servizio di salvataggio degli output degli algoritmi su database. Come semplificazione, è stato assunto che si volessero salvare tutti i dati di tutti gli algoritmi in esecuzione. Il compito del DBCoordinator, oltre a gestire la connessione al database MongoDB tramite gli oggetti della libreria MongoDBDriver, è quello di ricevere costantemente la lista di CameraManager dal Supervisor e di creare un attore DBWriter per ogni nuovo CameraManager rilevato. Un DBWriter è un semplice GenericClient implementato per rimanere sintonizzato sempre sulla stessa camera e salvare gli output sul database attraverso il proprio Sink (ovvero la funzione di consumo dei dati in uscita dal CameraManager/processo C++). Sotto il profilo del DDD questo attore è visto come entità, dato che necessita di due campi privati che ne costituiscono lo stato: una mappa che tenga traccia di tutti i CameraManager di cui ha già creato il rispettivo Saver e l'oggetto di MongoDBDriver che concretizza la connessione al database.

3.2.6 CLIClient

L'attore CLIClient implementa il GenericClient per stampare su console a riga di comando l'output del CameraManager su cui è sintonizzato. Utilizza un CLIParser per interpretare le stringhe dello standard input come comandi da eseguire su richiesta dell'utente (o di qualunque altra entità software che utilizzi l'applicazione in questo modo).

3.2.7 Server

Il Server rappresenta il componente di integrazione tra il sistema ad attori e il frontend in React. Questo modulo si sviluppa in due livelli per separare la gestione delle API HTTP dalla logica di elaborazione dei dati provenienti dai CameraManager.

3.2.7.1 VertxRouter

Il componente VertxRouter implementa il layer di comunicazione HTTP utilizzando il framework Vert.x, una soluzione reattiva e ad alte prestazioni adatta per applicazioni real-time. Tale scelta deriva dalla sua natura event-driven che si allinea con il paradigma ad attori utilizzato nel resto del sistema.

La classe mantiene uno stato interno che riflette lo stato corrente del sistema di videosorveglianza attraverso diversi campi: **currentCameraId** traccia quale telecamera è attualmente selezionata nell'interfaccia utente e **cameraMap** mantiene l'associazione tra identificatori stringa delle telecamere e le loro ActorRef nel sistema Akka - mapping importante per tradurre le richieste HTTP provenienti dal frontend in messaggi appropriati per il sistema ad attori. I campi relativi allo stato dei servizi come **subscribeStatus**, **inputStatus** e **configStatus** forniscono un meccanismo di monitoraggio dello stato del sistema che viene esposto attraverso l'endpoint **/status**.

L'endpoint REST **/camera/switch** gestisce il cambio di telecamera attiva non limitandosi a modificare lo stato locale, ma propagando il cambiamento al sistema ad attori

attraverso il messaggio di **SwitchToCamera**, permettendo così a tutti i componenti del sistema di rimanere sincronizzati. L'endpoint **/window**, invece, gestisce la configurazione delle regioni di interesse per l'analisi video: quando riceve le coordinate di una finestra di selezione, il router valida i parametri ricevuti (verificando che le coordinate non siano negative e che le dimensioni siano positive) e costruisce anche una mappa strutturata che viene poi inoltrata al CameraManager appropriato attraverso un messaggio **ForwardConfigData**.

3.2.7.2 GUIBackEnd

Il GUIBackEnd rappresenta l'implementazione concreta di un **GenericClient** specializzato per l'integrazione con l'interfaccia web.

La funzione **startingSinkFunction** definisce il comportamento di consumo dei dati provenienti dallo stream del CameraManager. In particolare, l'implementazione effettua il parsing dei dati ricevuti estraendo le metriche di rilevamento - come il numero di persone rilevate, la modalità dell'algoritmo che si sta utilizzando e i frame al secondo - e li propaga al VertxRouter per l'esposizione attraverso le API REST. Questa pipeline di elaborazione include una gestione degli errori che previene il crash del sistema in caso di dati malformati.

Il metodo **onMessage** sovrascrive il comportamento astratto per gestire i messaggi specifici del contesto GUI. La gestione del messaggio **CameraMap** è particolarmente importante in quanto stabilisce il collegamento bidirezionale tra il VertxRouter e il sistema ad attori, permettendo al router di inviare comandi al sistema. I messaggi di successo e fallimento dei vari servizi (SubscribeServiceSuccess, InputServiceFailure, ecc.) vengono tradotti in aggiornamenti di stato che l'interfaccia web può interrogare attraverso l'endpoint **/status** e fornendo un feedback visivo all'utente sullo stato delle operazioni.

L'elaborazione del messaggio **ForwardConfigData** serve a gestire la configurazione del processo C++: quando viene ricevuta una richiesta di configurazione con una nuova finestra di selezione definita dall'utente, il GUIBackEnd costruisce dinamicamente la linea di comando per il processo C++ includendo i parametri della finestra (se presenti). La costruzione del comando include il percorso completo dell'eseguibile e tutti i parametri necessari, garantendo che il processo C++ venga riavviato con la configurazione corretta.

La combinazione di questi due componenti (VertxRouter e GUIBackend) fornisce un'intermediazione tra l'interfaccia React del frontend e il sistema di elaborazione video basato su attori, garantendo che le metriche di rilevamento siano disponibili in tempo reale all'interfaccia utente mantenendo al tempo stesso la flessibilità utile per future estensioni del sistema.

3.3 Deployment

Il deployment dell'applicazione avviene tramite tecniche di containerizzazione e orchestrazione che verranno spiegate in seguito. Lo scopo di questo capitolo è spiegare i ruoli all'interno del dominio che i vari tipi di nodi hanno. Grazie ad Akka (sia Cluster che Stream), è possibile ignorare completamente la distribuzione delle JVM all'interno della rete, rendendo trasparente all'applicazione la gestione dei container Docker da parte dell'orchestrazione.

In questo contesto i nodi sono file Docker Compose che accorpano servizi rappresentati dai singoli container e possono essere visti come l'insieme di programmi eseguiti su una singola macchina fisica. I nodi sono stati concepiti per specializzare il carico di lavoro della macchina che li ospita, in modo da ottimizzare il software e ottenere prestazioni accettabili anche nel caso in cui l'algoritmo di CV sia computazionalmente oneroso.

3.3.1 User Node

Il nodo utente è l'insieme di container dispiegato sulla macchina fisica dell'utilizzatore dell'applicativo: vi risiedono le due interfacce (grafica e a riga di comando) da cui è possibile interagire con il resto del sistema. Ogni nodo utente rappresenta un singolo client (o due, nel caso in cui entrambe le interfacce siano utilizzate contemporaneamente) e pertanto si suppone che nel caso d'uso medio il numero di nodi di questo tipo sia al contempo molto alto e altamente variabile nel tempo: per questo motivo è stato previsto che la connessione o la disconnessione di ogni nodo utente possa avvenire in qualunque momento, senza ripercussioni sul resto del sistema e senza requisiti temporali particolari.

3.3.2 Utility Node

Il nodo utilità contiene tutti i componenti attivi e passivi di cui è necessaria una sola istanza, ovvero il database, gli attori che lo gestiscono e il Supervisor.

Nello scenario di utilizzo medio esiste una sola istanza attiva di questo nodo, con eventuali altre istanze di backup pronte a entrare in funzione in caso di partizionamento della rete.

3.3.3 Camera Node

Questo nodo viene dispiegato su ogni macchina (presumibilmente on-edge e dotata di hardware specializzato nell'esecuzione parallela su GPU) a cui è fisicamente collegata una camera. In questo modo si mantiene il carico di lavoro dovuto all'elaborazione dell'algoritmo di CV separato da quello relativo alla gestione di tutti gli attori Scala tranne CameraManager, necessario per propagare l'output dell'algoritmo verso i client.

Nel tipico caso d'uso si avranno tanti nodi camera quante camere saranno presenti nel sistema.

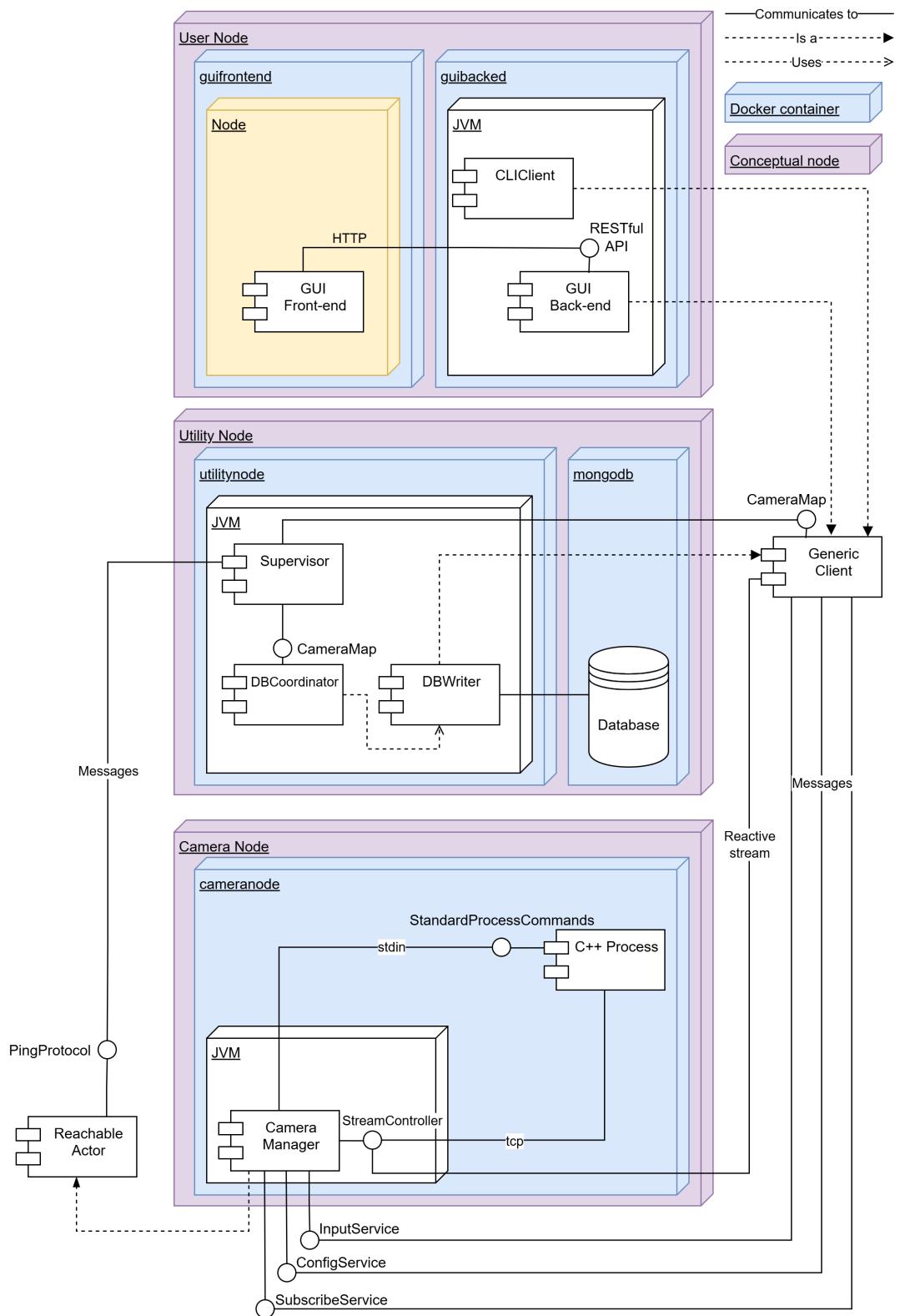


Figura 5: Diagramma UML di deployment del sistema.

4 Dettagli implementativi

4.1 Strumenti di Akka utilizzati

4.1.1 Receptionist

Akka Receptionist è un modulo del framework Akka(?), progettato per semplificare la gestione delle richieste di servizio e la distribuzione di informazioni tra attori all'interno di un sistema basato su Akka.

In sostanza, Akka Receptionist fornisce un'implementazione pronta all'uso di un modello di servizio di registrazione e ricerca. Permette agli attori di registrarsi come fornitori di servizi e di cercare servizi registrati da altri attori all'interno del sistema.

Quando un attore desidera fornire un servizio, può registrarsi presso il Receptionist fornendo una chiave univoca associata al servizio. Allo stesso modo, quando un attore desidera utilizzare un servizio, può cercarlo nel Receptionist specificando la chiave corrispondente al servizio di interesse.

Il Receptionist funge da intermediario, mantenendo un registro dei servizi registrati e consentendo agli attori di richiederli in modo efficiente.

4.1.2 Cluster

Per quanto riguarda la creazione del Cluster e dei relativi componenti, è stato necessario definire programmaticamente all'interno di *Builder* la loro configurazione. In particolare, è stata considerata una configurazione di base (*initBasicConfig*) - in modo tale da inizializzare il cluster - che permette lo scambio dei messaggi sulla rete tramite il protocollo TCP, la loro relativa serializzazione mediante il serializzatore di Akka *jackson-json* e la simulazione di più nodi fisici distribuiti nella stessa JVM.

In seguito, a partire da questa, sono state implementate due diverse estensioni: una relativa alla semplice aggiunta di ulteriori nodi all'interno del cluster (*setConfig*) e l'altra per la definizione e l'aggiunta dei cluster singleton (*setClusterConfig*).

Dunque, la creazione del Cluster avviene attraverso la creazione di ActorSystem aventi ciascuno una delle suddette configurazioni, a seconda del ruolo che devono assumere all'interno dell'architettura del sistema.

4.1.3 Stream

Dopo svariati tentativi, si è giunti alla conclusione che il paradigma ad attori non fosse adeguato al throughput richiesto per la diffusione dell'output di un programma di CV in tempo reale. Per questo motivo si è scelto di utilizzare il paradigma reattivo per la comunicazione tra CameraManager e GenericClient, implementato dalla libreria Akka Stream.

4.2 Applicazione di Computer Vision

Il file definito in `domain/src/main/cpp` rappresenta un programma di video-sorveglianza che combina computer vision, streaming in tempo reale e comunicazione di rete. La sua struttura segue un pattern modulare dove ogni componente ha responsabilità specifiche ma collabora strettamente con gli altri attraverso meccanismi di sincronizzazione avanzati e protocolli di comunicazione diversi.

4.2.1 Flusso Principale dei Dati

Il sistema opera secondo una pipeline che parte dalla cattura del video e arriva fino alla distribuzione ai client. Inizialmente, il programma acquisisce frame video da una sorgente (webcam o file video) attraverso le API di OpenCV. Ogni frame catturato viene immediatamente processato dal sistema di rilevamento che identifica volti o corpi umani utilizzando algoritmi di machine learning pre-addestrati.

Una volta completata la detection, il frame viene arricchito con rettangoli verdi che evidenziano le persone rilevate. Questo frame elaborato viene poi compresso in formato JPEG e distribuito simultaneamente a tutti i client connessi tramite WebSocket. Parallelamente, le informazioni statistiche sulla detection (numero di persone rilevate, FPS, modalità di rilevamento) vengono inviate a un sistema di gestione centralizzato chiamato CameraManager (application layer).

4.2.2 Componenti Principali

La classe Detector rappresenta il cervello del sistema di computer vision poiché incapsula due diversi algoritmi di rilevamento: HOG (Histogram of Oriented Gradients) per il rilevamento del corpo umano intero e Haar Cascades per il rilevamento specifico dei volti. La particolarità di questa implementazione consiste nella possibilità di cambiare dinamicamente tra le due modalità e nella capacità di operare su regioni specifiche del frame piuttosto che sull'intera immagine, ottimizzando così le performance computazionali.

La classe VideoServer gestisce l'intero sistema di comunicazione WebSocket: non si limita a trasmettere video, ma implementa un vero e proprio server concorrente capace di gestire connessioni multiple simultanee. Utilizza la libreria websocketpp per creare un server asincrono che può servire più client contemporaneamente senza degradare troppo le performance. La gestione delle connessioni avviene attraverso meccanismi thread-safe che garantiscono la stabilità del sistema anche con carico più elevato.

4.2.3 Architettura Concorrente e Gestione Thread-Safe

L'aspetto più importante del programma è la sua natura multi-threaded con sincronizzazione basata su primitive di C++. Il sistema implementa tre thread principali che operano in modo coordinato attraverso meccanismi di sincronizzazione specifici.

Il thread principale gestisce il server WebSocket e l'accettazione delle connessioni utilizzando il pattern asincrono di websocketpp basato su **Boost::Asio**. La gestione delle

connessioni attive è protetta da un `std::mutex connectionsMutex` che garantisce accesso esclusivo alla struttura dati `std::set<connection_hdl>` durante le operazioni di inserimento ed eliminazione delle connessioni client.

Il thread secondario si occupa della pipeline video completa, dall'acquisizione frame alla distribuzione. La coordinazione tra i thread avviene attraverso una variabile `std::atomic<bool> running` che permette la terminazione sicura senza race condition ed evita l'uso di mutex pesanti per il semplice controllo di stato.

Il terzo thread gestisce la comunicazione con il CameraManager attraverso una socket TCP, implementando un listener non-bloccante con timeout configurabile in modo che il sistema riesca a rimanere responsivo anche quando la comunicazione di rete subisce latenze o interruzioni.

4.2.4 Integrazione con Sistema di Gestione Centralizzato

Un elemento distintivo di questa architettura è l'integrazione con un CameraManager esterno (del layer application) attraverso socket TCP. Questo componente riceve dati "statistici" in tempo reale da ogni istanza del programma e può inviare comandi di controllo. Infatti, la comunicazione è bidirezionale: il programma invia periodicamente dati sulle detection effettuate, mentre il manager può inviare comandi speciali (come il carattere 'k' per far terminare il programma di visione) per coordinare l'intero sistema di telecamere.

Il protocollo di comunicazione implementa un meccanismo di reconnection automatica con retry a intervalli fissi di 2 secondi tra ogni tentativo, permettendo al sistema di mantenersi operativo anche in caso di interruzioni temporanee della rete. L'architettura distribuita generale del progetto permette di scalare il sistema facilmente, aggiungendo nuove telecamere che si registrano automaticamente presso il manager centrale, creando così una rete di sorveglianza coordinata dove ogni nodo può operare autonomamente ma contribuisce a una visione d'insieme del sistema di monitoraggio.

4.3 Applicazione Web: Frontend

L'applicazione web implementata è utilizzata per la gestione e il monitoraggio di flussi video provenienti da telecamere. L'architettura è costruita utilizzando React come framework di frontend, implementando un pattern di comunicazione real-time attraverso WebSocket per lo streaming video e chiamate REST per il controllo del sistema.

4.3.1 Componente Principale

Il file `App.js` in `interface/client/src` rappresenta il componente principale che implementa una strategia di gestione centralizzata che coordina l'interazione tra i vari sottosistemi dell'applicazione.

Il componente traccia la telecamera attualmente selezionata, mantenendo un elenco aggiornato di tutte le telecamere disponibili nel sistema (componente "Camera Controls" nella figura 6). Parallelamente, gestisce metriche operative che includono il conteggio

delle persone rilevate, la modalità operativa corrente dell'algoritmo usato e i frame per secondo elaborati (componenti "People Count", "Mode" e "Frame per second" nella figura 6).

Un aspetto importante dell'implementazione è il meccanismo di polling periodico, implementato attraverso `useEffect`, con cui ogni 5 secondi l'applicazione interroga l'endpoint `/status` del server per ottenere aggiornamenti sullo stato del sistema. Questo garantisce che l'interfaccia utente rimanga sincronizzata con lo stato reale del backend, anche in presenza di eventi asincroni o modifiche esterne al sistema, ed è implementato attraverso una gestione degli errori che verifica sia il codice di risposta HTTP e sia il content-type - per garantire l'integrità dei dati ricevuti.

La gestione del cambio telecamera, invece, rappresenta un altro aspetto importante del componente poiché quando l'utente seleziona una nuova telecamera, il sistema non si limita ad aggiornare l'interfaccia locale, ma notifica attivamente il backend attraverso una chiamata POST all'endpoint `/camera/switch` garantendo così che il backend possa adeguare i propri processi di elaborazione video alla nuova sorgente selezionata. Inoltre, il componente utilizza una chiave di riconnessione (`reconnectKey`) che forza il componente `VideoFeed` a reinizializzarsi completamente quando cambia la telecamera, ottenendo una transizione più pulita tra i flussi video (tasto "Restart Camera" nella figura 6).

4.3.2 Sistema di Controllo Telecamere

Il componente **CameraList** implementa l'interfaccia di controllo per la gestione delle telecamere disponibili nel sistema. Questo modulo non permette soltanto la visualizzazione di una lista, ma offre funzionalità di controllo e configurazione integrate con il resto dell'architettura layered. L'aspetto visuale del componente rappresenta ogni telecamera con un pulsante che mostra l'id e il nome della videocamera (figura 6). La telecamera attualmente attiva viene evidenziata attraverso stili CSS e un badge "Active", rendendo chiaro qual è il flusso video attualmente visualizzato.

La funzionalità di configurazione funziona in modo tale che quando l'utente richiede il riavvio di una telecamera attraverso il pulsante "Restart Camera", il sistema invia una richiesta POST all'endpoint `/window` con l'identificativo della telecamera. Questa operazione trigerà sul backend un processo di riconfigurazione che può includere il riavvio del processo di acquisizione video o la modifica dei parametri di streaming. Durante questa operazione, il componente gestisce lo stato dell'interfaccia disabilitando il pulsante e mostrando un'interfaccia di caricamento.

4.3.3 Sistema di Streaming Video

VideoFeed rappresenta il componente tecnicamente più complesso dell'applicazione poiché gestisce lo streaming real-time del video attraverso WebSocket e implementa la funzionalità di selezione di regioni di frame per l'analisi video.

La gestione della connessione WebSocket è stata implementata con particolare attenzione alla resilienza: il sistema implementa un meccanismo di riconnessione automatica con

retry esponenziale, tentando fino a 10 riconnessioni, in modo da prevenire il sovraccarico del server in caso di problemi temporanei di rete e garantire il ripristino automatico del servizio quando possibile. In particolare, il componente mantiene traccia del tempo trascorso dall'ultimo frame ricevuto e mostra un overlay di riconnessione solo dopo un ritardo significativo, evitando di notificare l'utente per interruzioni momentanee.

Per quanto riguarda l'elaborazione dei frame video ricevuti attraverso WebSocket, ogni messaggio contiene dati binari in formato JPEG che vengono convertiti in Blob e la pipeline di elaborazione è ottimizzata per minimizzare l'overhead di memoria attraverso la pulizia immediata degli URL oggetto/Blob dopo il rendering, per prevenire problemi di memoria. Il componente traccia anche le dimensioni reali del frame per garantire che le operazioni di selezione della regione mantengano la precisione indipendentemente dal ridimensionamento dell'interfaccia.

Nello specifico, la funzionalità di selezione della regione permette all'utente di definire un'area di interesse direttamente sul feed video attraverso un'interfaccia drag-and-drop (figura 7). Il sistema gestisce questa funzionalità con due modalità operative: la selezione del punto iniziale e la definizione della regione attraverso trascinamento. Le coordinate vengono convertite dalle coordinate del canvas visualizzato alle coordinate reali del frame video, garantendo precisione nell'elaborazione backend. Dunque, quando una regione viene selezionata e inviata al server, il componente applica automaticamente un fattore di correzione per compensare il ridimensionamento applicato dal backend e permettere così la propagazione delle giuste dimensioni verso l'algoritmo.

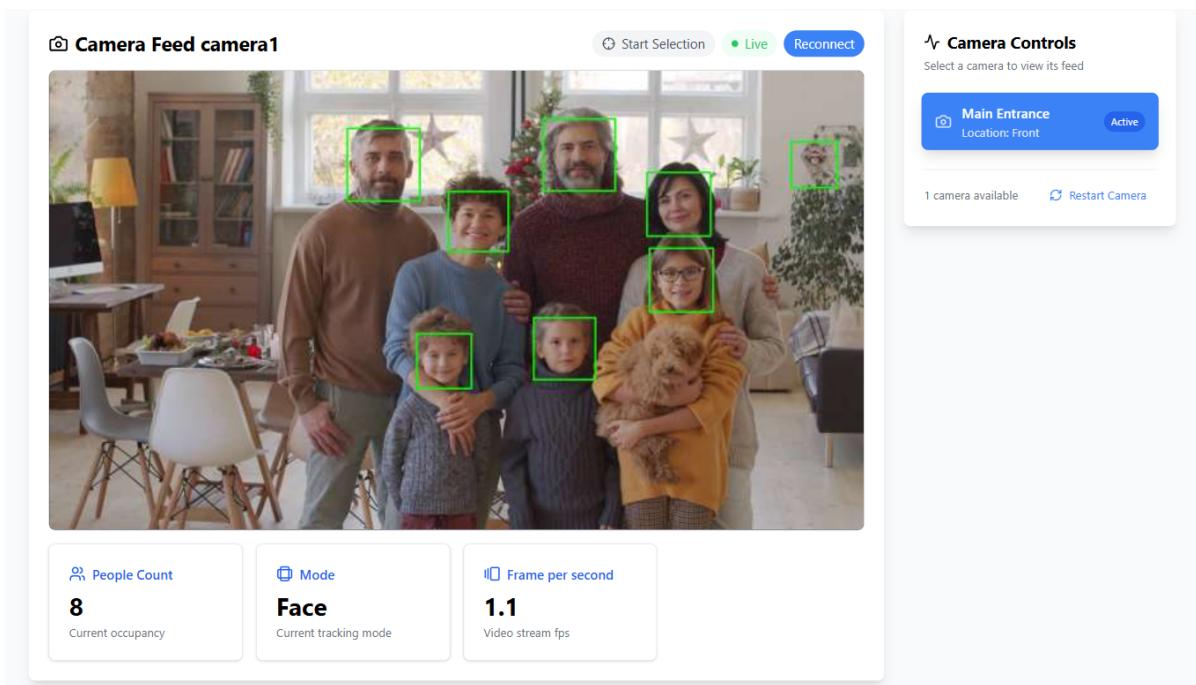


Figura 6: Interfaccia grafica frontend React

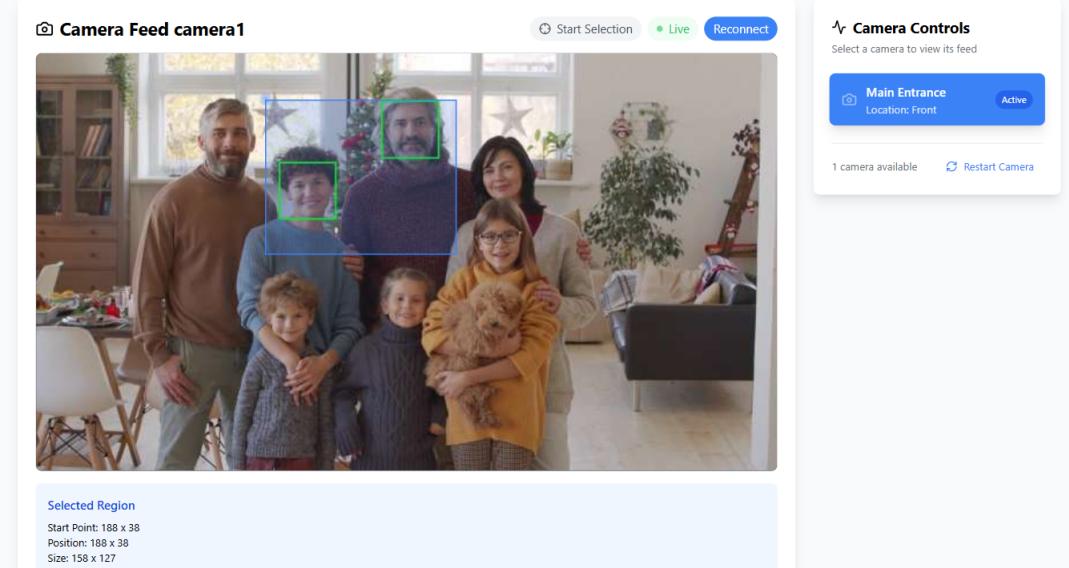


Figura 7: Funzionalità di selezione regione di interesse

4.4 Deployment e Configurazione del Sistema ad Attori Distribuito

Il sistema distribuito è costruito su tre tipologie di nodi specializzati che formano un cluster Akka, ciascuno con responsabilità definite per il proprio ruolo. Questa architettura serve ad ottenere una scalabilità orizzontale e fault tolerant attraverso la distribuzione dei componenti.

4.4.1 Entry Point delle Applicazioni

Il file **App.scala** di ciascun nodo (in *distribution/{nomenodo}/src/main*) rappresenta il punto di ingresso dell'applicazione di ciascun nodo e implementa la logica di inizializzazione del cluster Akka. Questi entry point seguono un pattern comune ma con specializzazioni specifiche per ogni ruolo nel sistema.

Il **CameraNode** - con porta 2553 nel cluster Akka - inizializza un ActorSystem con un CameraManager come attore root, passando la porta 8080 per la comunicazione con il processo C++ di elaborazione video. La configurazione programmatica sovrascrive i parametri di default specificando l'hostname canonico "cameranode" e configurando i seed nodes del cluster per il discovery iniziale. L'utilizzo di ConfigFactory.parseMap permette di costruire dinamicamente la configurazione fondendo i parametri di runtime con quelli statici definiti nei file di configurazione (situati in *distribution/cameranode/src/main/resources*).

Lo **UserNode** - con porta 2551 nel cluster Akka - , che ospita il GUIBackEnd, segue una struttura simile ma inizializza l'attore Server che gestisce l'integrazione con l'interfaccia web. La configurazione include parametri specifici per la gestione delle connessioni HTTP e WebSocket, oltre alle impostazioni del cluster. L'hostname "guibackend" viene utilizzato per identificare univocamente questo nodo nel cluster, facilitando il routing dei messaggi e il discovery dei servizi.

L'**UtilityNode** - con porta 2552 nel cluster Akka - presenta una configurazione più particolare poiché implementa il pattern Cluster Singleton per garantire che esistano esattamente un'istanza del Supervisor e una del DBCoordinator nell'intero cluster. Il Supervisor viene creato con una strategia di supervisione che prevede il resume in caso di errore, garantendo continuità operativa anche in presenza di eccezioni non fatali. In generale, questo design è fondamentale per mantenere la consistenza dello stato globale del sistema e coordinare le operazioni distribuite.

4.4.2 Configurazione del Sistema Akka

La configurazione di questo sistema si sviluppa su due livelli: **reference.conf** che fornisce i default dell'applicazione e **application.conf** che contiene le configurazioni specifiche del deployment - entrambi situati in *distribution/{nomenodo}/src/main/resources* di ciascun nodo. Questa separazione segue le best practise di Akka ed è stata necessaria per permettere gli override selettivi mantenendo una base configurativa stabile.

5 DevOps

La strategia DevOps implementata nel progetto DCCV si articola attraverso quattro elementi tecnologici principali e interconnessi che lavorano sinergicamente per garantire un ciclo di vita del software robusto e scalabile (build automation, containerization, orchestration e continuous integration in figura 8).

- Il primo elemento, la **build automation**, utilizza Gradle come orchestratore centrale per gestire la complessità multi-tecnologica del sistema, coordinando la compilazione di componenti C++ attraverso CMake, la gestione di dipendenze Scala/Akka e l'integrazione del frontend in React;
- Il secondo elemento, la **containerizzazione**, trasforma ciò che è stato prodotto dalla build automation in unità di deployment auto contenute attraverso configurazioni Docker specializzate per ogni layer dell'architettura;
- Il terzo elemento, l'**orchestrazione**, coordina questi container in un cluster distribuito utilizzando Docker Swarm per abilitare la scalabilità, la resilienza e la gestione automatizzata del deployment;
- Il quarto elemento, la **continuous integration**, automatizza l'intero processo di sviluppo attraverso GitHub Actions che gestiscono la propagazione dei commit tra i branch dell'architettura del sistema, il versionamento semantico automatico e la generazione di release complete con artefatti di deployment - come fat JAR, immagini Docker e script di orchestrazione (figura 8).

L'integrazione tra questi elementi principali segue una progressione in cui ogni fase abilita e ottimizza la successiva: la build automation produce artefatti standardizzati che la containerizzazione incapsula in ambienti riproducibili; l'orchestrazione li distribuisce e gestisce come un cluster distribuito coeso; e la continuous integration automatizza e governa l'intero flusso dalla modifica del codice al deployment in produzione. Questa architettura permette di concentrarsi sulla logica applicativa mentre l'automazione gestisce la complessità operativa, dalla compilazione al deployment distribuito, garantendo qualità attraverso fasi automatizzate di testing e consistenza architetturale attraverso vincoli procedurali che riflettono la struttura DDD del sistema.

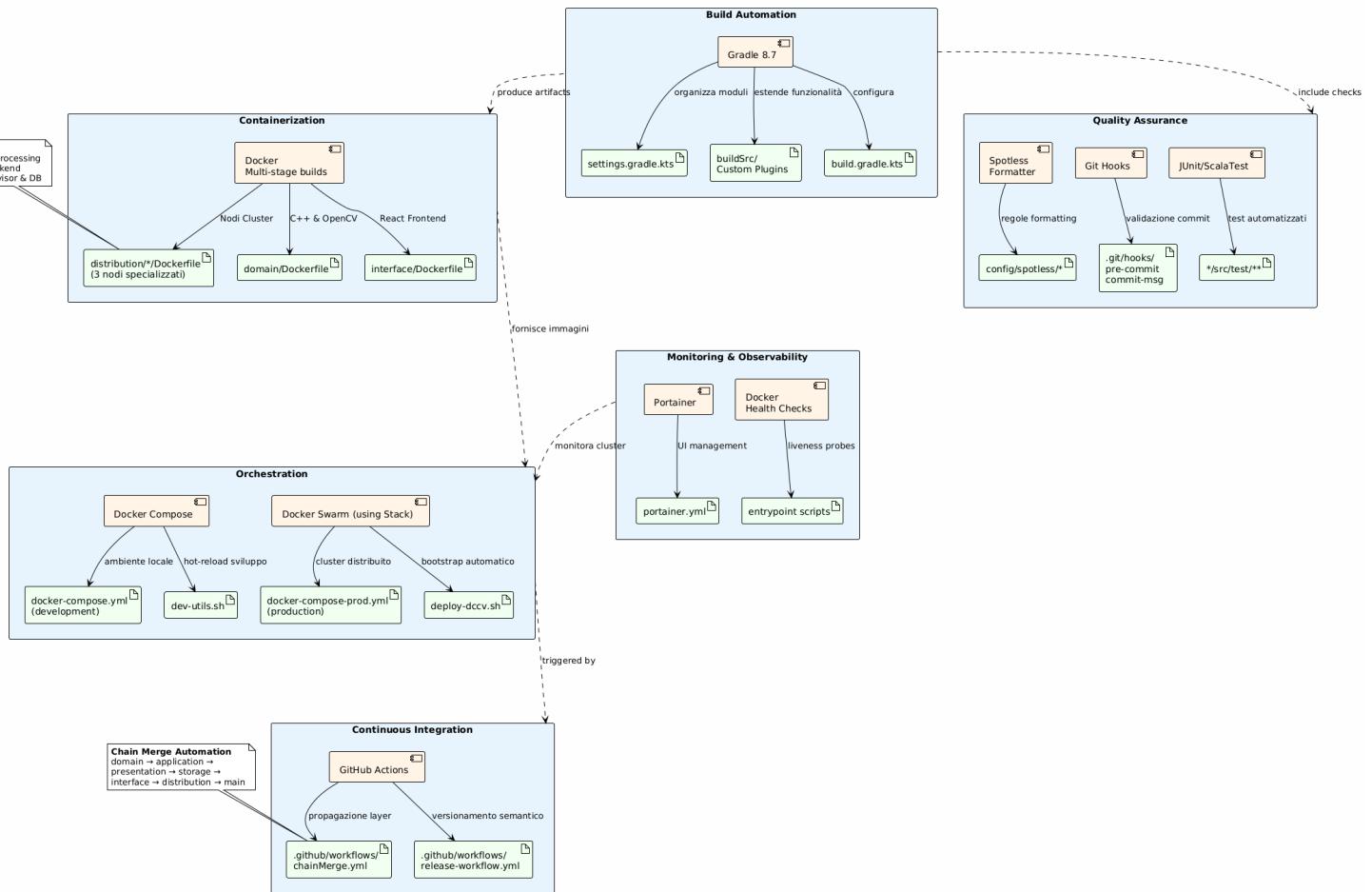


Figura 8: DevOps Overview: concetti DevOps → tecnologie → implementazioni

5.1 Build automation

In questa sezione seguirà la descrizione dettagliata della build automation relativa ad ogni livello dell'architettura. Lo scopo è quello di presentare e motivare ciascuna implementazione evidenziando i meccanismi più importanti.

5.1.1 Domain Layer

Questo livello dell'architettura contiene il programma applicativo principale che consiste in un algoritmo di visione artificiale implementato in C++ - tramite la libreria opencv -, e che utilizza la comunicazione via websocket per effettuare lo stream continuo sia del video elaborato e sia dei relativi dati come il numero di volti rilevati, la modalità di funzionamento dell'algoritmo e il numero di fps elaborati.

Per gestire in maniera più semplice e flessibile le dipendenze del programma C++ con le

varie librerie necessarie, è stato definito un file CMakeLists che configura il percorso sia per trovare OpenCV in base al sistema operativo, sia per le librerie necessarie (OpenCV, WebSocket++, Boost), e genera i Makefiles contenenti le istruzioni specifiche per compilare e linkare il programma. Quando il sistema operativo non è UNIX-compatibile, la configurazione indirizza esplicitamente lo sviluppatore verso i task gradle disponibili che a tale scopo utilizzano Docker. Così, come spiegato successivamente e poi nel paragrafo 5.2, la strategia implementa la containerizzazione come meccanismo di astrazione per normalizzare l'ambiente di build, privilegiando la riproducibilità attraverso Docker rispetto alla ricerca di una portabilità a livello di codice sorgente.

Il processo di compilazione del programma C++ avviene attraverso una pipeline di build ibrida orchestrata da Gradle (figura 9), dove il task centrale `buildCMake` esegue in sequenza i comandi CMake - come `cmake -build .` e `cmake -install .` - per generare, compilare e installare il programma C++.

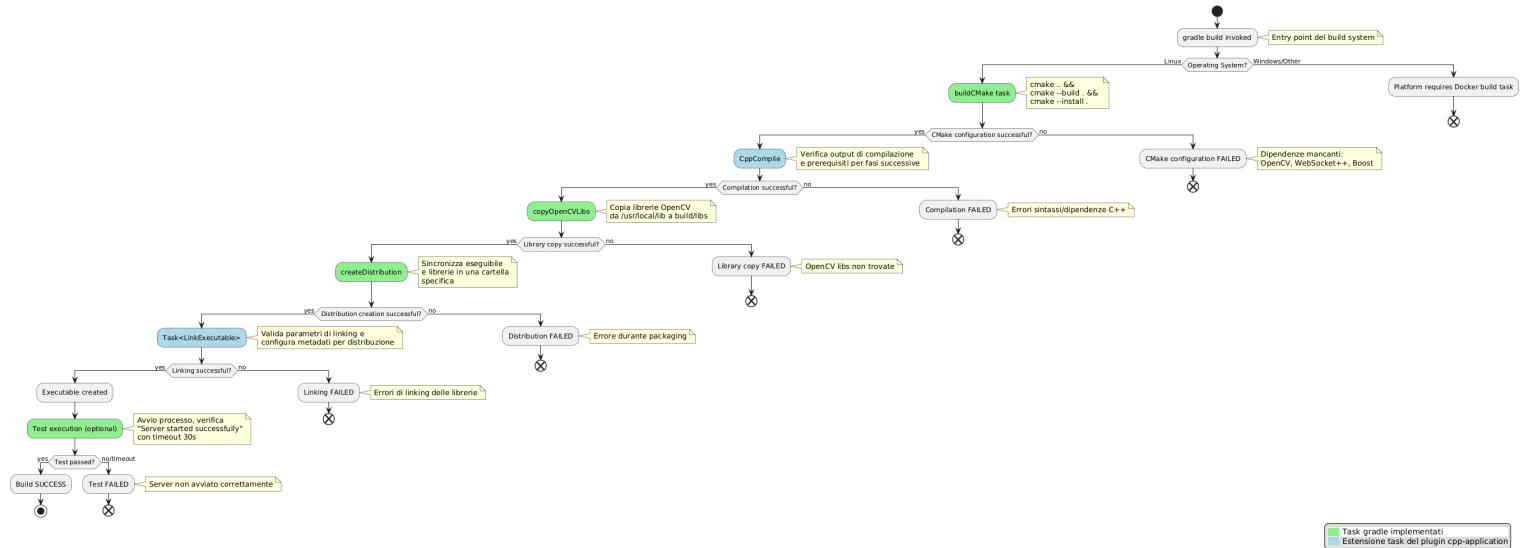


Figura 9: Activity-diagram del processo di build della compilazione C++

I task `CppCompile` e `LinkExecutable` del plugin **cpp-application** dipendono dal completamento di `buildCMake` e assumono ruoli di validazione e coordinamento nel processo: il primo verifica che la compilazione eseguita da CMake abbia prodotto gli output attesi e che tutti i prerequisiti per le fasi successive siano soddisfatti; il secondo valida che i parametri di linking configurati (come `-lopencv_*` libraries e configurazioni `-Wl,-rpath`) siano stati applicati correttamente e prepara i metadati necessari per la distribuzione finale dell'eseguibile.

In particolare, il task `LinkExecutable` dipende anche dal completamento dei task `copyOpenCVlibs` e `createDistribution`, i quali si occupano rispettivamente di copiare le librerie OpenCV necessarie da `/usr/local/lib` alla directory locale `build/libs`

del progetto, creando una distribuzione self-contained che elimina la dipendenza da installazioni di sistema specifiche; sincronizzare tutti i componenti (eseguibile compilato, librerie copiate e script di avvio) nella struttura unificata *build/release/domain*, producendo un package completo pronto per la distribuzione. Questa strategia permette di ottenere automaticamente tutti gli artefatti necessari per l'esecuzione dell'applicazione C++ in un formato che facilita lo sviluppo, il testing e la distribuzione senza richiedere configurazioni manuali dell'ambiente target.

Completato il processo di packaging, il sistema esegue un task `test` (fig. 9) di integrazione end-to-end che avvia effettivamente l'eseguibile compilato e monitora il suo output per verificare che il server riesca ad avviarsi correttamente entro un timeout (30 secondi), implementando così una forma di build verification testing che valida non solo la correttezza della compilazione e del linking, ma anche la capacità dell'applicazione di inizializzare correttamente tutte le sue dipendenze e componenti in un ambiente reale di esecuzione.

Come accennato all'inizio di questo paragrafo, nel caso in cui il sistema operativo sottostante non è UNIX-compatibile, sono stati implementati dei task che usano Docker e che è possibile usare per riuscire ugualmente a compilare ed eseguire il codice C++ presente. Il task principale `dockerBuild` automatizza completamente il processo di containerizzazione attraverso una pipeline in due fasi: prima costruisce automaticamente un'immagine Docker che incapsula tutte le dipendenze necessarie per la compilazione, poi esegue il container configurando il volume mounting per creare una connessione bidirezionale tra la directory di build locale e quella del container. Durante l'esecuzione containerizzata, il sistema invoca automaticamente la stessa catena di task Gradle analizzata precedentemente (figura 9), ma all'interno dell'ambiente Docker normalizzato, eliminando qualsiasi dipendenza da configurazioni specifiche del sistema host. La configurazione del task gestisce le differenze tra sistemi operativi host (Windows e Unix-like) adattando i comandi shell appropriati, mentre il mapping specifico della directory di build permette di ottenere direttamente nell'ambiente host tutti i file risultanti dalla compilazione del programma C++ (eseguibili, librerie, artefatti di distribuzione) senza necessità di operazioni aggiuntive. In questo modo è stato possibile trasformare un processo potenzialmente complesso e error-prone in un'operazione del tutto automatizzata che richiede un singolo comando (`./gradlew dockerBuild`), astraendo la complessità della gestione di ambienti di sviluppo eterogenei.

5.1.2 Application Layer

Il layer application implementa una strategia di build automation completamente diversa rispetto al modulo domain, riflettendo la transizione dall'ambiente C++ all'ecosistema JVM e specificatamente alle tecnologie Scala per sistemi distribuiti. Questa separazione architettonica riflette una strategia di design che ottimizza ciascun modulo per responsabilità specifiche: mentre il modulo domain gestisce computazione intensiva e integrazione con librerie native, il modulo application si concentra sull'orchestrazione distribuita e la coordinazione di componenti attraverso message passing asincrono. La

configurazione Gradle in *application/build.gradle.kts* orchestra un processo di build che gestisce dipendenze relative alla programmazione ad attori e ai sistemi reattivi: **Scala 3 library** (versione 3.3.3) come linguaggio principale che fornisce capacità di programmazione funzionale avanzate, **Akka Actor Typed** per l'implementazione del modello ad attori distribuiti che costituisce il core architettonico dell'orchestrazione del sistema, **RxScala** per la gestione di stream reattivi e programmazione asincrona, e un ecosistema di testing che combina **ScalaTest** con **Akka TestKit** per supportare i test di sistemi concorrenti e distribuiti attraverso simulazione.

La configurazione del build system pone l'attenzione alla gestione delle versioni e alla compatibilità, utilizzando il Version Catalog attraverso **gradle/libs.versions.toml** per centralizzare la gestione delle dipendenze e garantire consistenza tra diverse librerie dell'ecosistema Akka che hanno interdipendenze precise e dove versioni incompatibili possono causare problemi sottili di runtime difficili da diagnosticare. Il sistema di testing implementato affronta il testing di sistemi ad attori - che differiscono molto da quello sequenziale tradizionale perché devono gestire comportamenti asincroni, message ordering non deterministico, scenari di fault tolerance, ecc. - configurando JUnit Platform Engine per supportare ScalaTest e integrando Akka Actor TestKit che fornisce strumenti specializzati per controllare i messaggi, simulare scenari di failure e altro. L'integrazione di Scala 3 toolchain con Java 17 permette di sfruttare le advanced features di Scala 3 (come improved type inference e miglior pattern matching) mantenendo al contempo accesso completo all'intero ecosistema Java esistente e alle sue librerie mature.

In questo caso, dunque, la build automation gestisce automaticamente la complessità della compilazione incrementale di Scala, la risoluzione delle dipendenze transitive nell'ecosistema Akka e l'integrazione con IDE moderni, creando un workflow automatizzato che facilita lo sviluppo di sistemi distribuiti.

5.1.3 Presentation Layer

Il layer presentation dimostra come tramite la build automation è possibile anche gestire efficacemente la composizione di dipendenze tra moduli in progetti articolati. Nel nostro caso, la configurazione Gradle in *presentation/build.gradle.kts* implementa una strategia di dependency composition attraverso la dichiarazione **IMPLEMENTATION(PROJECT(":APPLICATION"))**, che stabilisce una dipendenza diretta dal modulo application permettendo l'ereditarietà automatica di tutto l'ecosistema tecnologico sottostante (Akka Actor Typed, Scala 3, ScalaTest con Akka TestKit) senza necessità di ridichiarazione esplicita delle versioni o configurazioni. Questa strategia evita la duplicazione di configurazioni e garantisce consistenza nella gestione delle dipendenze transitive, permettendo al modulo presentation di concentrarsi esclusivamente sulle proprie responsabilità specifiche: l'implementazione del GenericClient che fornisce un'astrazione riutilizzabile per diversi tipi di client che devono interagire con il sistema distribuito, e del Supervisor che mantiene una vista globale dello stato dell'intero sistema attraverso pattern di monitoraggio distribuito.

Così facendo, la build automation orchestra automaticamente la risoluzione delle dipendenze inter-modulo e l'ordine di compilazione, garantendo che modifiche nei moduli sottostanti triggereranno la ricompilazione incrementale dei moduli dipendenti. Il sistema

di testing, invece, eredita tutta l'infrastruttura necessaria per verificare comportamenti che coinvolgono interazioni cross-layer.

Dunque, questa architettura di build sfrutta la dependency composition per ottenere un sistema modulare scalabile in cui ogni layer aggiunge funzionalità specifiche senza compromettere la coerenza tecnologica e la manutenibilità del progetto, permettendo così lo sviluppo parallelo su diversi layer con integrazione automatica garantita dal sistema di build.

5.1.4 Storage Layer

Il layer storage introduce nella build automation la gestione di dipendenze per l'integrazione con sistemi di persistenza esterni. Estendendo la strategia di composizione multi-layer già vista nel presentation layer, la build configuration in *storage/build.gradle.kts* implementa una doppia dipendenza attraverso `IMPLEMENTATION(PROJECT(":APPLICATION"))` e `IMPLEMENTATION(PROJECT(":PRESENTATION"))`, creando una catena di dipendenze transitive più complessa che il sistema di build deve orchestrare automaticamente. Questa configurazione permette l'ereditarietà automatica dell'intero stack tecnologico sottostante mentre introduce nuove dipendenze specializzate: **MongoDB Driver Sync** (versione 4.8.2) per la connettività database e la libreria **BSON** per la serializzazione documenti. Il sistema di build gestisce automaticamente la risoluzione delle compatibilità tra queste nuove dipendenze database e l'ecosistema Akka preesistente, prevenendo conflitti nel classpath e garantendo che l'ordine di compilazione rispetti la gerarchia di dipendenze tra i tre layer.

5.1.5 Interface Layer

Il layer interface è il primo che introduce la gestione di tecnologie eterogenee e l'orchestrazione di sistemi multi-linguaggio all'interno di un singolo layer. La configurazione principale in *interface/build.gradle.kts* implementa una strategia di aggregazione che coordina tre sottoprogetti distinti: **cli**, **server** e **client**, ciascuno con le proprie specificità tecnologiche e requisiti di build automation (figura 10).

In particolare, la build automation per questo layer gestisce simultaneamente ecosistemi di build completamente diversi che devono interoperare. I sottoprogetti **cli** e **server** condividono entrambi la strategia di dependency composition multi-layer vista nei moduli precedenti, dichiarando dipendenze sia da APPLICATION che da PRESENTATION per ereditare l'intero ecosistema Scala/Akka sottostante. Tuttavia, ciascuno estende questo stack tecnologico per scopi specifici: il modulo cli aggiunge funzionalità per l'interfaccia a riga di comando, mentre il modulo server introduce una complessità aggiuntiva a livello di dipendenze attraverso l'integrazione di **Vert.x** per la gestione HTTP e REST API, insieme alle dipendenze MongoDB per la persistenza (vedi figura 10).

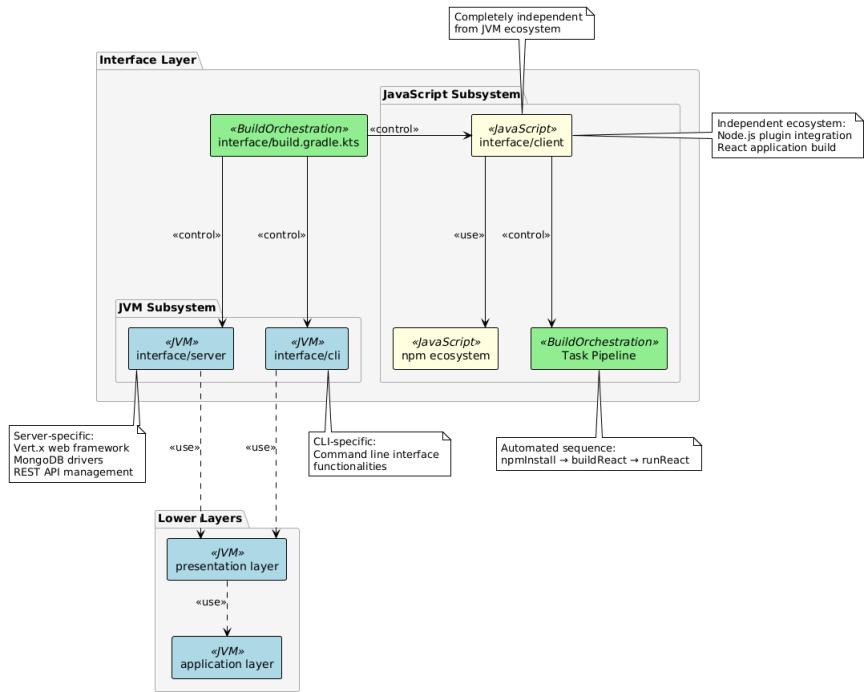


Figura 10: Component-diagram della build automation del layer interface

La complessità maggiore emerge nel sottoprogetto **client**, dove la build automation deve gestire un salto tecnologico dall'ecosistema JVM a quello JavaScript/Node.js. La configurazione in *interface/client/build.gradle.kts* utilizza il plugin **com.github.node-gradle.node** per integrare npm e Node.js all'interno del workflow Gradle, creando un ponte tra due mondi di build automation tradizionalmente separati. Il sistema di build configura una catena di dipendenze automatica dove il task **BUILDREACT** dipende da **NPMINSTALL** e il task **RUNREACT** dipende da **BUILDREACT** (figura 10), garantendo che l'installazione delle dipendenze npm, la compilazione dell'applicazione React e l'esecuzione del development server avvengano nell'ordine corretto senza intervento manuale.

5.1.6 Distribution Layer

Il layer distribution è quello in cui tutte le tecnologie e i pattern implementati nei layer precedenti convergono per creare un sistema di packaging e deployment distribuiti. La configurazione della build automation per questo layer gestisce la creazione di nodi specializzati che incapsulano selettivamente funzionalità da tutti i layer sottostanti, implementando strategie di containerizzazione e clustering che rendono il sistema distribuito sviluppato pronto per il deployment.

La struttura modulare del layer distribution - visibile nel file *settings.gradle.kts* attraverso i sottomoduli **:DISTRIBUTION:UTILITYNODE**, **:DISTRIBUTION:USERNODE** e **:DISTRIBUTION:CAMERANODE** -, richiede alla build automation di gestire configurazioni di di-

pendenze specializzate per ciascun tipo di nodo. Ogni sottoprogetto implementa una strategia di dependency composition selettiva che eredita solo i componenti necessari dallo stack tecnologico sottostante: il CameraNode include il layer domain per le funzionalità C++/OpenCV insieme ai componenti application per la gestione degli attori CameraManager; l'UserNode compone i layer presentation e interface, con focus sui componenti server, per l'esposizione di API REST tramite Vert.x; l'UtilityNode integra i layer storage, presentation e application per fornire funzionalità di supervisione e persistenza distribuite (vedi figura 11).

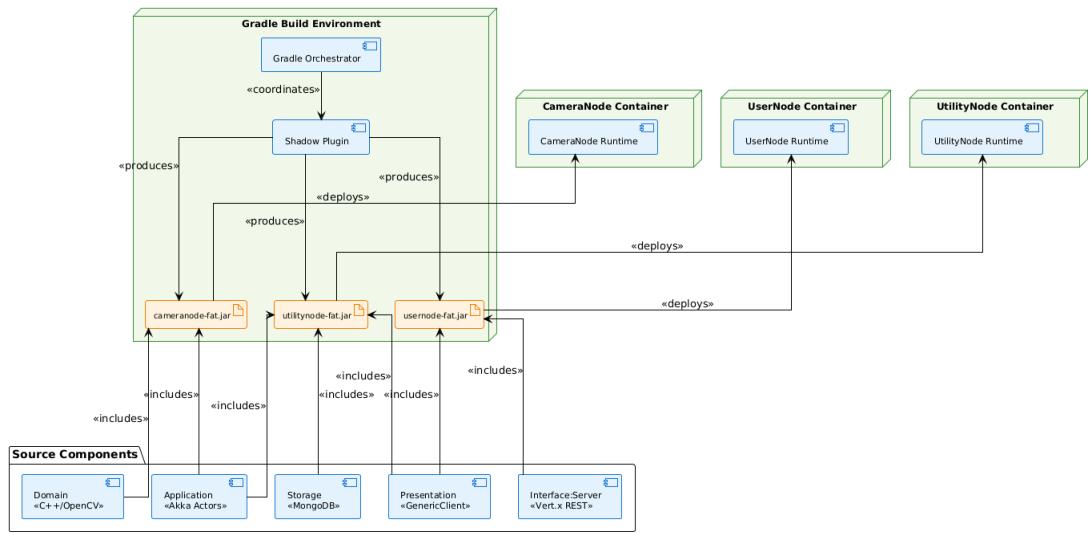


Figura 11: Deployment-diagram della build automation del layer distribution

Un elemento distintivo in questo layer è l'utilizzo del plugin **com.github.johnrengelman.shadow**, che trasforma le dipendenze transitive in artefatti di distribuzione self-contained. La configurazione tramite il task `shadowJar` in ciascun sottoprogetto implementa la creazione di JAR autocontenuti (figura 11) che gestiscono automaticamente la fusione di file tramite `mergeServiceFiles()`, l'eliminazione di firme digitali incompatibili e l'include dei file fondamentali della configurazione di Akka (`reference.conf`, dichiarati in `distribution/{nomenodo}/src/main/resources` di ciascun sottoprogetto del layer) tramite *Appending Transformer*.

Occorre precisare che un elemento tecnico fondamentale che abilita la creazione di fat JAR funzionanti per sistemi ad attori distribuiti è proprio la corretta configurazione dei file Akka presenti in `distribution/nomenodo/src/main/resources`. La definizione dei file **application.conf** e **reference.conf** rappresenta una delle sfide più complesse nell'implementazione del sistema, poiché questi file controllano aspetti critici come la configurazione dei seed nodes, le porte di clustering, i timeout di connessione, i meccanismi di failure detection specifici per ogni tipo di nodo e tanto altro. Senza una configurazione precisa di questi parametri, i fat JAR generati dal task `shadowJar` sarebbero tecnicamente validi ma incapaci di formare un cluster Akka, rendendo inefficace l'intera

strategia di distribuzione. La complessità deriva dal fatto che ogni nodo deve avere una configurazione che bilanci l'autonomia locale con la capacità di integrarsi dinamicamente in un cluster distribuito, richiedendo parametrizzazioni specifiche per porte, indirizzi IP, strategie di clustering ecc. che devono rimanere coerenti attraverso ambienti diversi.

Tornando alla build automation, essa gestisce automaticamente l'ordine di compilazione cross-layer garantendo che quando viene invocato il task `shadowJar` di un nodo, tutti i layer dipendenti siano stati preventivamente compilati e i loro artefatti siano disponibili per l'inclusione nel fat JAR finale. Questa strategia risolve una delle sfide più complesse riscontrate nella distribuzione di sistemi ad attori: garantire che tutte le configurazioni, le dipendenze e i metadati necessari per il clustering Akka siano correttamente inclusi in un singolo artefatto distribuibile.

Infine, il file `distribution/build.gradle.kts` implementa il task di deployment che fornisce l'automazione per il deployment dello stack Docker Swarm con Portainer per il monitoring del cluster (la relativa spiegazione è approfondita nei paragrafi 5.2 e 5.3).

5.1.7 Quality Automation e Process Governance

Parallelamente al sistema di build automation, il progetto implementa una strategia di quality automation attraverso Git hooks e strumenti di code formatting che garantiscono consistenza e qualità a livello di processo di sviluppo. Il sistema è orchestrato attraverso due Git hooks principali: il `PRE-COMMIT` hook che esegue automaticamente il task `checkPreCommit` - il quale a sua volta invoca `spotlessCheck` e `spotlessApply` per verificare la conformità del codice e applicare il formatting automatico -, e il `COMMIT-MSG` hook che implementa la validazione dei Conventional Commits attraverso il task `conventionalCommits` che utilizza un pattern regex per imporre la struttura standardizzata dei messaggi di commit. Il nucleo del sistema di formattazione è rappresentato dal plugin **Spotless**, configurato in `buildSrc/src/main/kotlin/dccv-spotless.gradle.kts`, che gestisce automaticamente la formattazione di più tecnologie utilizzate nel progetto: C++, utilizzando CLANG-FORMAT con stile Google per garantire consistenza con le best practice; Scala attraverso SCALAFMT, configurato tramite il file dedicato `config/spotless/scalafmt.conf`, per seguire le convenzioni della community; Kotlin Gradle scripts con KTLINT, che legge le impostazioni da `config/spotless/.editorconfig`; JavaScript tramite PRETTIER, configurato attraverso `config/spotless/prettier.json`, per consistenza cross-platform; e JSON con GSON formatter per una struttura uniforme.

La strategia di gestione delle configurazioni esterne implementata attraverso file dedicati nella directory `config/spotless/` non solo centralizza le policy di formattazione rendendo il sistema facilmente manutenibile ed evolvibile, ma crea anche integrazione con l'IDE attraverso il file `.editorconfig` che standardizza le impostazioni di editing e elimina la disconnessione tra automazione automatica e esperienza interattiva dello sviluppatore. Un altro aspetto rilevante del sistema è la gestione automatica delle dipendenze dei formatter attraverso il task `downloadFormatterBinaries`, che scarica e configura automaticamente tutti gli strumenti necessari (clang-format, ktlint, scalafmt) con gestione condizionale basata sul sistema operativo, eliminando la necessità di installazioni manuali complesse e garantendo che tutti (gli sviluppatori) utilizzino esattamente le

stesse versioni degli strumenti. La configurazione include meccanismi di ottimizzazione avanzati come il **ratcheting** che applica le regole di formattazione solo ai file modificati rispetto al branch di riferimento, permettendo l'introduzione graduale di standard di qualità in progetti esistenti senza richiedere un refactoring eccessivo, e strategie di caching che preservano le performance durante l'esecuzione ripetuta dei controlli di qualità.

Questa implementazione trasforma processi tradizionalmente manuali in automazione che lavora senza ostacolare lo sviluppatore, mentre contemporaneamente impone standard di qualità che facilitano la code review, riducono il "rumore" nei commit e migliorano la manutenibilità complessiva del codebase. In questo modo è stato possibile ottenere un sistema coordinato di controlli di qualità che agisce sia sul codice che sui commit, supportando un processo di sviluppo che combina velocità di iterazione con alti standard qualitativi, e che si integra con il sistema di build automation.

5.2 Containerization

L'applicazione DCCV sfrutta la containerizzazione basata su Docker per garantire portabilità, isolamento e riproducibilità dell'ambiente di esecuzione attraverso tutti i nodi dell'architettura distribuita. Questa scelta è stata fatta per poter gestire un sistema complesso che integra tecnologie eterogenee - dalla logica applicativa in Scala alle librerie native C++ per l'elaborazione delle immagini, fino alle interfacce web in React.

In particolare vi sono tre layer principali che sfruttano Docker. Il **layer distribution** costituisce il nucleo della containerizzazione poiché implementa tutti i Dockerfile specializzati per ogni tipologia di nodo, gli script di entrypoint per la gestione del ciclo di vita dei container e le configurazioni necessarie per l'effettivo deployment distribuito. Ogni Dockerfile in questo layer implementa un approccio multi-stage build che ottimizza la dimensione finale delle immagini separando nettamente la fase di compilazione da quella di runtime e riducendo l'overhead operativo.

Il **layer interface** presenta una particolare integrazione con la containerizzazione attraverso il packaging del frontend React e la gestione degli endpoint di comunicazione tra i diversi servizi. Inoltre, gli script di entrypoint implementano una logica per la gestione dello stato dei container e la preparazione dell'ambiente di runtime, includendo controlli di integrità e meccanismi di recovery automatico.

Il **layer domain** sfrutta la containerizzazione per la compilazione e l'integrazione delle componenti native C++ necessarie per l'elaborazione video in tempo reale. Durante la fase di build dei container, il sistema compila automaticamente le librerie OpenCV e le dipendenze native, garantendo che ogni container disponga delle risorse computazionali necessarie per l'esecuzione delle operazioni di computer vision.

Infine, ogni tipologia di nodo (CameraNode, UserNode e UtilityNode) viene incapsulata in immagini Docker specializzate che includono solo le dipendenze strettamente necessarie per la propria funzione specifica, seguendo il principio di responsabilità singola anche a livello di container. Questa separazione facilita la manutenzione, il debugging e l'evoluzione indipendente di ogni componente del sistema.

5.2.1 Domain Layer

Il layer domain rappresenta il componente più critico e tecnicamente complesso dell'architettura DCCV dal punto di vista della containerizzazione, poiché deve garantire un ambiente robusto e riproducibile per l'esecuzione di algoritmi di computer vision ad alte prestazioni.

La strategia di containerizzazione del domain layer si basa su una soluzione custom rappresentata dall'immagine **brunoesposito2/ubuntu_opencv_build**, che abbiamo sviluppato specificamente per risolvere un problema relativo all'ecosistema OpenCV: l'assenza di immagini Docker ufficiali con versioni aggiornate di OpenCV e la limitata disponibilità di pacchetti recenti nei repository standard di Ubuntu. Abbiamo sviluppato una soluzione from-scratch compilando e installando l'ultima versione di OpenCV (la versione è la 4.11 ed era l'ultima rilasciata da opencv nel momento in cui stavamo lavorando a questa parte di progetto) direttamente in un'immagine Docker Ubuntu, per ottenere un ambiente riproducibile e ottimizzato che è stato successivamente pubblicato su DockerHub.

Questa immagine personalizzata rappresenta più di una semplice configurazione di dipendenze, poiché la compilazione statica di OpenCV all'interno dell'immagine Docker risolve contemporaneamente diversi problemi importanti. Primo, elimina completamente le dipendenze runtime esterne garantendo che l'applicazione sia completamente autosufficiente e immune alle variazioni dell'ambiente host. Secondo, assicura la riproducibilità dell'ambiente di elaborazione - aspetto cruciale in computer vision perché anche minime differenze nelle versioni delle librerie possono alterare i risultati degli algoritmi di detection, compromettendo l'accuratezza e la consistenza delle elaborazioni. Terzo, la disponibilità pubblica dell'immagine su DockerHub trasforma una soluzione specifica del progetto in una risorsa riutilizzabile che accelera lo sviluppo di futuri progetti basati su OpenCV.

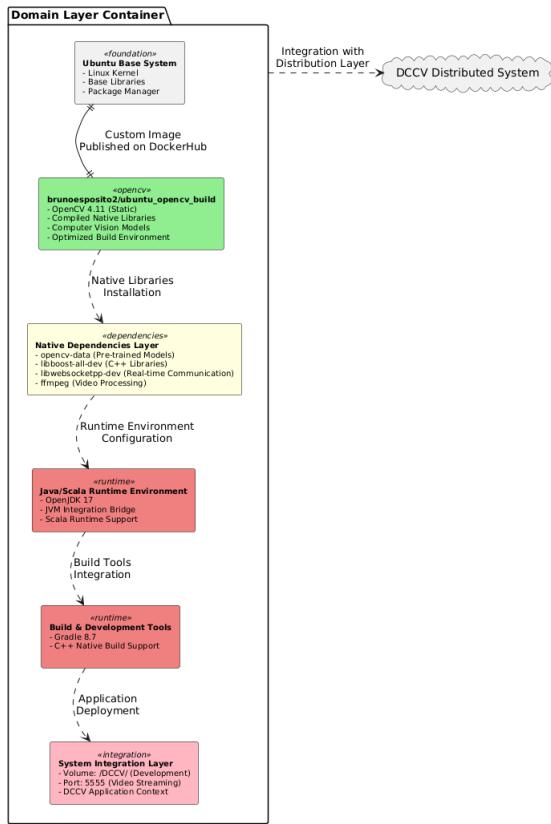


Figura 12: Component-diagram del Dockerfile del layer domain

Il Dockerfile del domain layer, oltre ad usare questa immagine personalizzata, effettua una sequenza di installazioni (figura 12). La prima fase installa le dipendenze native essenziali per l'elaborazione delle immagini: **opencv-data** fornisce i modelli pre-addestrati per la detection, **libboost** offre le librerie C++ avanzate per la gestione efficiente delle strutture dati e **libwebsocketpp** abilita la comunicazione real-time con i client web per lo streaming video. L'inclusione di **ffmpeg** permette l'eventuale conversione e manipolazione di flussi video in diversi formati.

La seconda fase configura l'ambiente JVM con OpenJDK 17, creando il ponte necessario tra il mondo C++ del domain layer e la logica applicativa Scala del resto del sistema tramite la configurazione delle variabili d'ambiente.

L'installazione di Gradle completa l'ambiente abilitando la compilazione dei componenti C++ direttamente all'interno del container. Questo approccio elimina le dipendenze dall'ambiente di build dell'host e garantisce che la compilazione del codice C++ avvenga in un ambiente controllato e riproducibile.

Infine, la gestione dei volumi espone la **directory di lavoro /DCCV/** per la condivisione di dati durante lo sviluppo, mentre la **porta 5555** consente lo streaming video in tempo reale verso i client esterni, supportando così tanto le esigenze di sviluppo quanto quelle

operative. Questo Dockerfile è specificamente progettato per supportare lo **sviluppo** e il **testing** degli algoritmi di computer vision all'interno del domain layer, mentre il deployment dell'applicazione completa viene gestito attraverso i Dockerfile specializzati presenti nel layer distribution.

5.2.2 Interface Layer: Frontend

Mentre il layer domain affronta le complessità dell'integrazione multi-tecnologica per la computer vision, il layer interface presenta sfide architetturali complementari ma distinte, focalizzate sull'integrazione di tecnologie web con l'orchestrazione distribuita.

Il layer interface rappresenta il punto di contatto diretto tra gli utenti finali e l'architettura distribuita del progetto, richiedendo un approccio di containerizzazione che deve ottimizzare simultaneamente l'esperienza utente, le prestazioni di rendering e l'integrazione con i servizi backend distribuiti. La complessità di questo layer consiste nella necessità di creare un ambiente che supporti efficacemente lo sviluppo di applicazioni React pur mantenendo la capacità di integrarsi dinamicamente con l'orchestrazione Docker Swarm dell'intero sistema.

La strategia di containerizzazione dell'interface layer si articola attraverso un **approccio multi-stage** che riflette le diverse fasi del ciclo di vita di un'applicazione web moderna (figura 13). Il **primo stage** crea un ambiente di build completo che integra Node.js per la gestione dell'ecosistema JavaScript, npm per la risoluzione delle dipendenze frontend e OpenJDK 17 e Gradle per mantenere la coerenza con l'architettura di build del resto del sistema. Il processo di build implementato nel container utilizza configurazioni npm ottimizzate per ambienti di produzione, con timeout estesi e meccanismi di retry che garantiscono una maggiore robustezza del processo di installazione delle dipendenze anche in ambienti con rete più instabili. Tale processo termina con il comando npm run build che, eseguendo react-scripts build, genera una versione ottimizzata dell'applicazione React che include bundling, minification e performance migliori.

Il **secondo stage** rappresenta l'ambiente di runtime, basato su **docker:dind** ma configurato per includere esclusivamente i componenti necessari per l'esecuzione. L'utilizzo del pacchetto **serve** come server web statico rappresenta una scelta volta a bilanciare semplicità operativa e prestazioni. Infatti, questo approccio elimina la complessità di configurare server web più pesanti - come Nginx o Apache - pur fornendo tutte le funzionalità necessarie per servire un'applicazione React con routing e gestione delle risorse statiche.

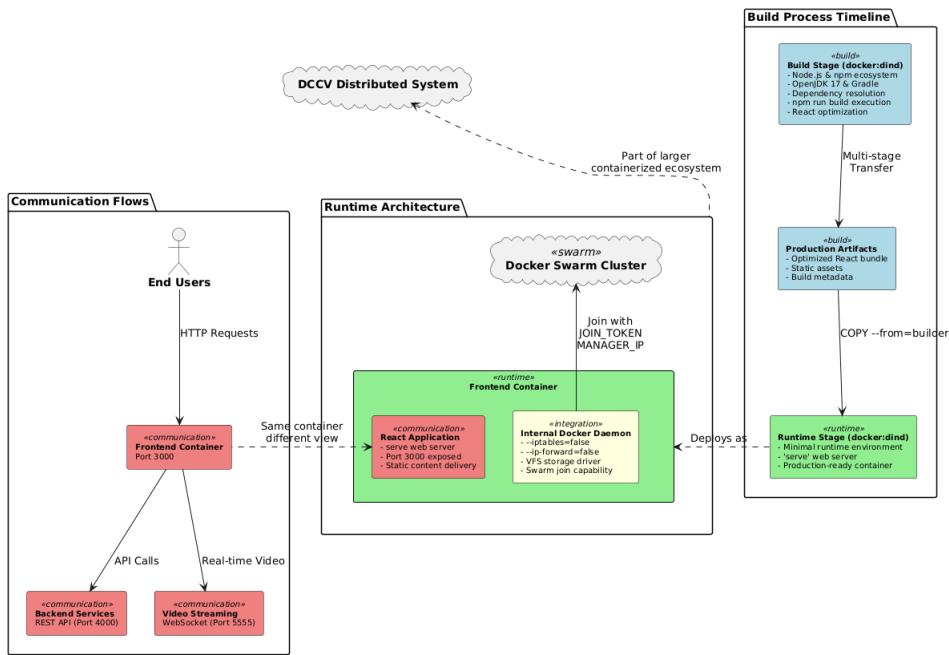


Figura 13: Multi-perspective architectural diagram del Dockerfile del layer interface

A rendere particolare la containerizzazione del layer interface è l'utilizzo della configurazione Docker-in-Docker - accennata poco fa - che risponde alla necessità dell'architettura di permettere al container di frontend di unirsi attivamente al Docker Swarm come nodo partecipante (figura 13, l'argomento viene approfondito in seguito al paragrafo 5.3). Lo script di entrypoint implementa configurazioni specifiche per l'ambiente frontend che includono sia una logica di join automatico al Docker Swarm attraverso i token di autenticazione e l'indirizzo IP del manager e sia una logica più elaborata per la gestione del daemon Docker interno che verifica lo stato dei processi esistenti, gestisce la pulizia delle risorse e configura parametri specifici per l'ambiente containerizzato come lo storage driver VFS e la disabilitazione di alcune delle funzionalità di rete avanzate - come la gestione automatica delle regole iptables e l'IP forwarding - che potrebbero creare conflitti con il networking overlay del Docker Swarm cluster. Le configurazioni `-iptables=false` e `-ip-forward=false` prevengono proprio i conflitti tra la gestione delle regole iptables del daemon Docker interno e quelle del daemon Docker del nodo manager che orchestra il cluster, evitando interferenze nel routing del traffico tra i nodi dello swarm.

Invece, l'esposizione della porta 3000 serve a permettere l'accesso esterno per gli utenti finali all'applicazione React del sistema.

Questo Dockerfile è stato progettato sia per supportare il **development** e **testing** dell'interfaccia utente all'interno del contesto dell'architettura distribuita e sia per essere utilizzato a **livello di rilascio**, fornendo così un ambiente che replica fedelmente le condizioni operative del sistema in produzione pur mantenendo la flessibilità necessaria per

l’iterazione rapida durante lo sviluppo del frontend.

5.2.3 Distribution Layer

Il layer distribution rappresenta il culmine dell’architettura di containerizzazione del progetto, poiché gestisce la convergenza di tutti i componenti sviluppati nei layer precedenti in una strategia di deployment unificata e ”production-ready”.

La complessità particolare di questo layer consiste nella necessità di standardizzare approcci di containerizzazione diversi, mantenendo al tempo stesso le specificità tecniche richieste da ciascun componente del sistema distribuito. A differenza dei layer precedenti che si concentravano su sfide tecnologiche specifiche, il distribution layer deve risolvere il problema dell’uniformità operativa attraverso la diversità implementativa.

Il metodo di containerizzazione di questo layer si basa su un pattern di standardizzazione **multi-stage**: tutti i Dockerfile del layer distribution implementano una separazione tra builder stage e runtime stage, in modo tale che la complessità di build non influenzi l’efficienza operativa dei container in esecuzione. Il **builder stage** standardizzato configura un ambiente completo che include Ubuntu come sistema base, OpenJDK 17 per la compatibilità JVM e Gradle 8.7 per la gestione del build, creando un layer di base riproducibile che garantisce consistenza di compilazione indipendentemente dall’ambiente host (figura 14).

In particolare, il processo di build utilizza il comando gradle shadowJar per produrre **fat JAR** che includono tutte le dipendenze necessarie per l’esecuzione autonoma di ciascun nodo, eliminando così le dipendenze runtime esterne, riducendo la complessità di deployment e permettendo a ogni container di operare senza richiedere configurazioni aggiuntive dell’ambiente host.

Invece, il **runtime stage** di ogni nodo implementa una configurazione che include OpenJDK 17 JRE per l’esecuzione dei fat JAR applicativi e Docker Engine per l’integrazione con Swarm. L’installazione di Docker all’interno dei container runtime abilita la configurazione Docker-in-Docker che permette a ogni nodo di partecipare attivamente al cluster Swarm, trasformando i container da semplici unità di deployment a partecipanti nell’orchestrazione distribuita (figura 14).

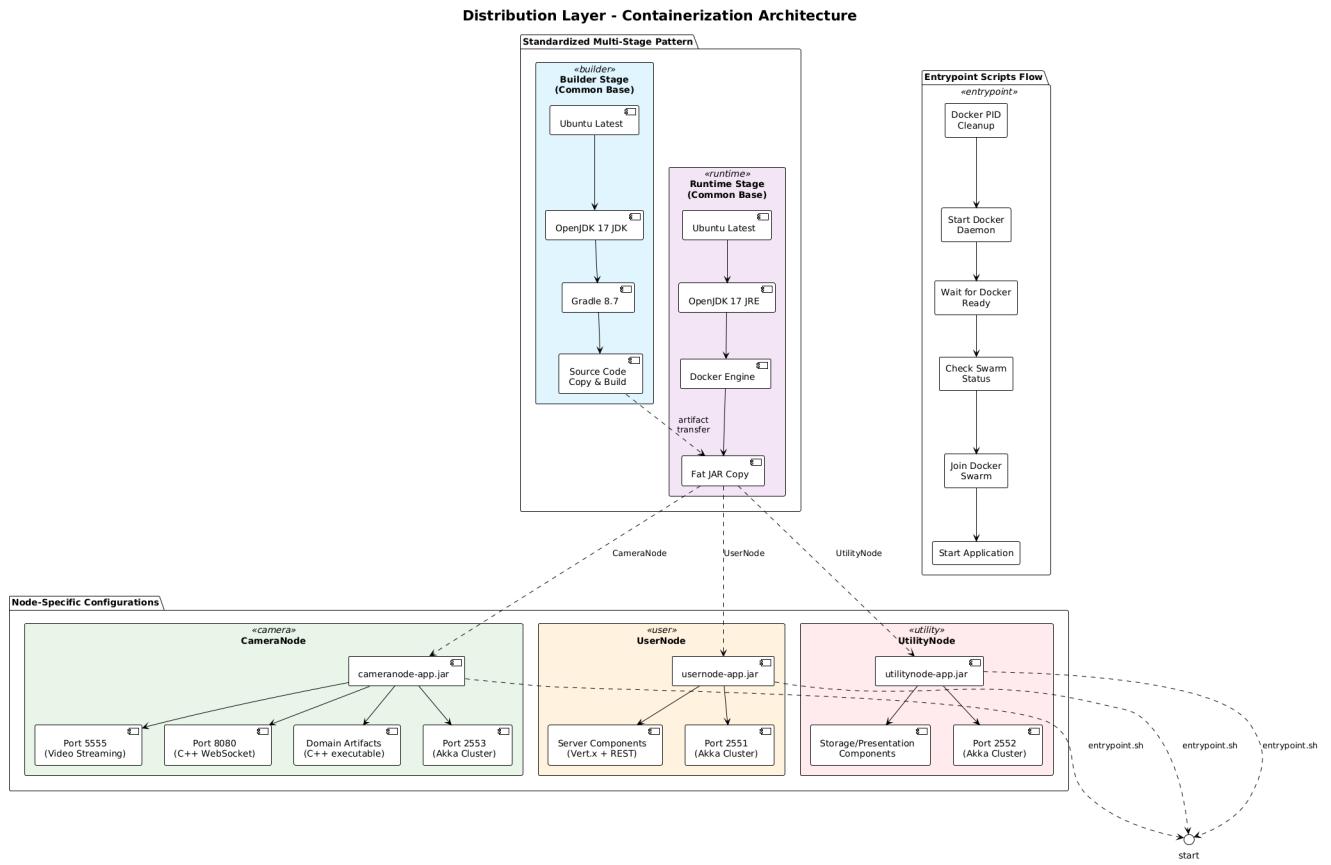


Figura 14: Component diagram dei Dockerfile del layer distribution

La configurazione delle porte esposte evidenzia l'architettura di comunicazione distribuita implementata nel sistema. Il CameraNode espone le porte 2553 per la comunicazione con gli altri nodi del cluster Akka, 5555 per lo streaming video e 8080 per la comunicazione via socket del programma C++ con il CameraManager; mentre UserNode e UtilityNode utilizzano rispettivamente le porte 2551 e 2552 per le comunicazioni inter-nodo del cluster Akka. In questo modo è stato possibile ottenere una separazione delle responsabilità di comunicazione implementando una topologia di rete che facilita la scalabilità orizzontale che il debugging operativo.

Un elemento importante della strategia di containerizzazione di questo layer è rappresentato anche dagli script di **entrypoint personalizzati** che gestiscono la logica di inizializzazione di ciascun nodo. Questi script implementano una sequenza di operazioni che include la gestione del daemon Docker interno, la pulizia di processi residui da precedenti esecuzioni, la configurazione dei parametri di rete specifici per l'ambiente containerizzato e l'esecuzione del join automatico al cluster Swarm utilizzando i token di autenticazione definibili come variabili globali (ad esempio tramite il file docker compo-

se, come spiegato nel paragrafo 5.3 successivo). Pertanto gli entrypoint rappresentano un layer di astrazione che nasconde la complessità dell'inizializzazione distribuita e che permette ai container di adattarsi dinamicamente all'ambiente di deployment e di integrarsi automaticamente nell'infrastruttura del cluster senza intervento manuale.

Questa standardizzazione della containerizzazione a livello di distribution layer crea le basi tecniche necessarie per implementare le strategie di orchestrazione che permettono al sistema DCCV di operare come un cluster distribuito coeso, argomento che verrà approfondito nella sezione successiva dedicata all'orchestrazione.

5.3 Orchestration

L'orchestrazione implementata nel progetto rappresenta l'evoluzione dell'architettura di containerizzazione sviluppata nel distribution layer, trasformando container autonomi in un unico cluster distribuito attraverso Docker Swarm. Questo risolve il problema di come coordinare l'esecuzione simultanea di componenti diversi che devono cooperare per fornire le funzionalità di computer vision distribuite, garantendo al tempo stesso resilienza, scalabilità e gestione automatizzata del ciclo di vita dei servizi.

La strategia implementata riflette una progressione dall'ambiente di sviluppo verso quello di produzione, utilizzando Docker Compose per la fase di development e Docker Stack (con Swarm) per il deployment di produzione. Questa separazione è utile perché mentre l'ambiente di sviluppo privilegia la velocità di iterazione e la facilità di debugging, l'ambiente di produzione ottimizza la robustezza, la scalabilità e la gestione automatizzata degli errori.

5.3.1 Ambiente di Sviluppo: Docker Compose e Automatizzazione Locale

L'orchestrazione per l'ambiente di sviluppo consiste in due componenti che lavorano insieme per semplificare il workflow: il file *distribution/docker-compose.yml* che definisce la topologia dei servizi e le loro interdipendenze, e lo script *distribution/dev-utils.sh* che fornisce un'interfaccia di automazione per le operazioni più comuni durante lo sviluppo. Il file docker-compose.yml bilancia la necessità di replicare l'architettura distribuita del sistema con l'esigenza di mantenere la flessibilità operativa richiesta durante lo sviluppo. La sua configurazione definisce cinque servizi principali che mappano direttamente i componenti architetturali del sistema: **cameranode** per l'elaborazione computer vision, **guifrontend** per l'interfaccia utente React, **guibackend** per i servizi REST e la coordinazione Akka, **utilitynode** per la supervisione del cluster e **mongodb** per la persistenza dei dati (vedi figura 15).

Un aspetto particolare di questa configurazione è l'implementazione della comunicazione inter-container attraverso la rete swarm-network che viene configurata come EXTERNAL: TRUE, permettendo ai container di development anche di integrarsi con un eventuale cluster Swarm già attivo sulla macchina di sviluppo. Questa scelta consente di testare localmente gli scenari di deployment distribuito senza compromettere la semplicità operativa dell'ambiente di development, permettendo una progressione da development semplice a testing distribuito attraverso le diverse modalità dello script **dev-utils.sh**.

approfondite in seguito.

La gestione delle variabili d'ambiente come JOIN_TOKEN e MANAGER_IP all'interno del docker-compose serve a preparare concettualmente il sistema per l'integrazione con Docker Swarm, anche quando viene eseguito in modalità standalone: ogni container riceve questi parametri che, sebbene non utilizzati direttamente o spesso in modalità development, garantiscono che gli entrypoint scripts e la logica applicativa rimangano consistenti tra ambiente di sviluppo e produzione.

La configurazione dei volumi persistenti per guifrontend-certs, guibackend-certs, utility-certs, cameranode-certs e mongo implementa una strategia di data management che preserva lo stato applicativo attraverso i restart dei container, cosa importante durante lo sviluppo quando modifiche al codice possono richiedere frequenti ricompilazioni e restart.

Lo script **dev-utils.sh** rappresenta un layer di astrazione che trasforma operazioni avanti diverse procedure in comandi semplici. Lo script fornisce diverse funzioni principali che coprono il ciclo di vita dello sviluppo: **start-dev** per l'avvio standard dei container di ciascun nodo; **start-dev-swarm** per il testing con Docker Swarm in locale; **sync-camera**, **sync-user**, **sync-utility** per la sincronizzazione selettiva dei nodi durante lo sviluppo, implementando un meccanismo di hot-reload che ricompila selettivamente solo i componenti modificati attraverso invocazioni Gradle specifiche (ad esempio `./gradlew application:jar domain:buildCMake distribution:cameranode:jar` per il CameraNode), seguito da restart soft delle applicazioni tramite `docker exec -it container pkill -f "java"` e un successivo riavvio automatico. Questo workflow appena descritto elimina la necessità di rebuild completi e permette cicli di development molto più rapidi.

Altre funzioni importanti dello script sono la **sync-all** per la sincronizzazione completa di tutti i nodi, **logs** per il monitoring real-time, **status** per verifiche di stato, **stop** per arresto controllato e **clean** per la pulizia completa dell'ambiente. In particolare, start-dev-swarm sfrutta la funzione **init_swarm** che esegue la sequenza di inizializzazione Docker Swarm attraverso verifiche di stato con `docker info --format '{{ .Swarm.LocalNodeState}}'`, inizializzazione condizionale - usando `docker swarm init` se lo swarm non è già stato creato -, creazione della rete overlay tramite `docker network create --driver overlay --attachable swarm-network` e configurazione delle variabili globali di join token. In questo modo si elimina la possibilità di errori che possono essere commessi manualmente e si garantisce un ambiente Swarm uguale e funzionante.

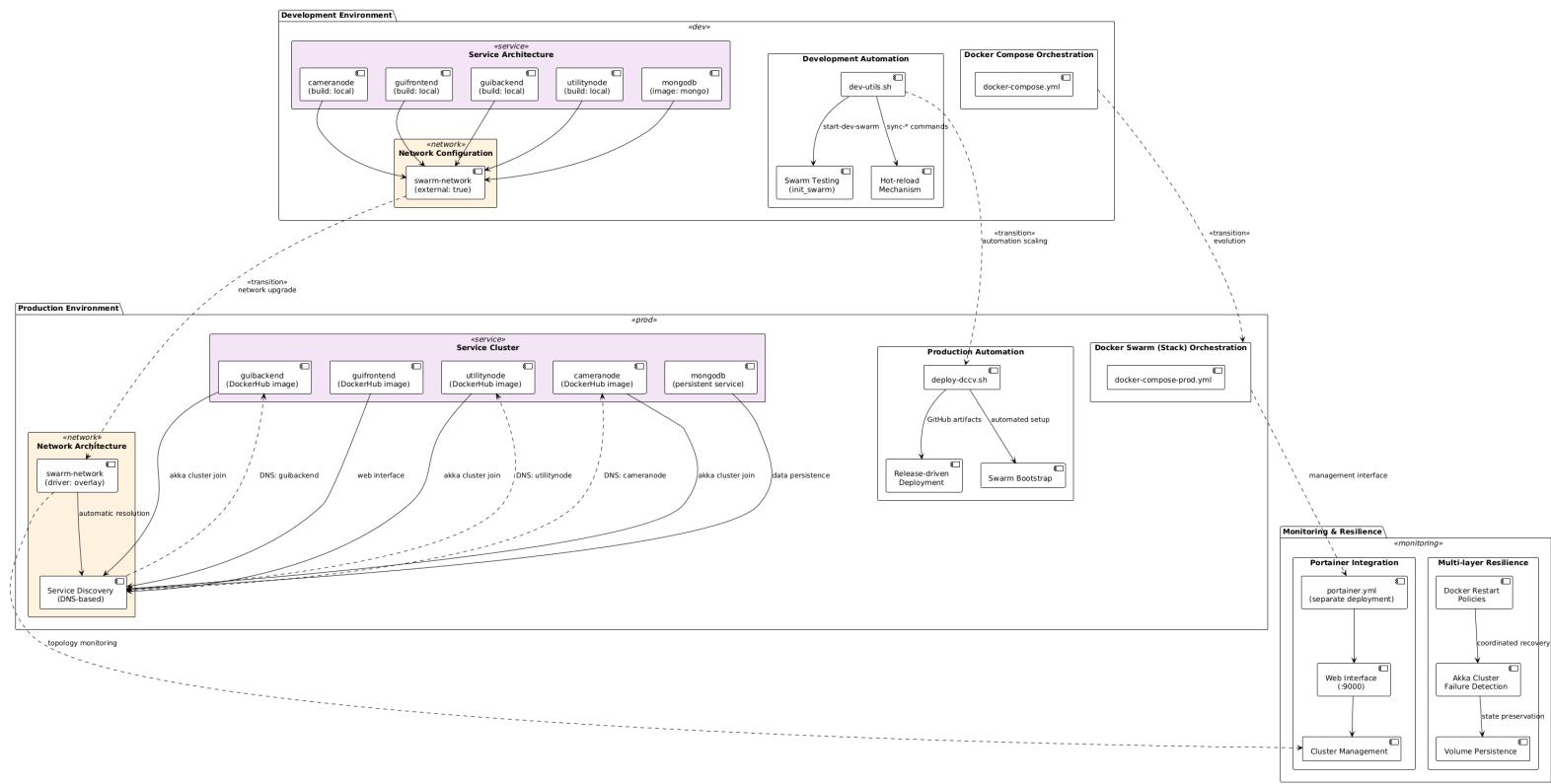


Figura 15: Architectural Transition diagram degli ambienti di sviluppo e produzione dell’orchestrazione

5.3.2 Ambiente di Produzione: Docker Swarm, Deployment Automatizzato e Monitoring

L’orchestrazione per l’ambiente di produzione implementa una strategia basata su Docker Swarm che trasforma il sistema da una collezione di container coordinati in un cluster distribuito auto-gestito. Questa transizione viene gestita attraverso tre componenti principali: il file `docker-compose-prod.yml` che definisce la topologia produttiva, lo script `deploy-dccv.sh` che automatizza il deployment completo e l’integrazione con **Portainer** per il monitoring e la gestione operativa.

5.3.2.1 Docker Swarm e Configurazione

Il file `docker-compose-prod.yml` implementa una versione evoluta del `docker-compose` di sviluppo, attraverso l’introduzione di configurazioni di deploy specifiche per Docker Swarm. Ogni servizio include ora istruzioni di `mode: replicated` e `restart_policy` con `condition: on-failure`, trasformando i container da semplici processi orchestrati in servizi Swarm gestiti automaticamente. La configurazione della rete passa da `external: true` a `driver: overlay` con `attachable: true`, creando

un’infrastruttura di comunicazione distribuita che permette ai nodi del cluster di comunicare attraverso più macchine (figura 15).

Una differenza fondamentale rispetto all’ambiente di sviluppo è l’utilizzo di immagini Docker pre-compilate e pubblicate su DockerHub (**brunoesposito2/dccv-cameranode:latest**, **brunoesposito2/dccv-guifrontend:latest**, **brunoesposito2/dccv-guibackend:latest** e **brunoesposito2/dccv-utilitynode:latest**, che verranno trattate successivamente nel paragrafo 5.5) invece di build locali (vedi figura 15). Questa scelta elimina completamente le dipendenze dall’ambiente di build locale e garantisce una riproducibilità assoluta del deployment attraverso ambienti diversi.

La configurazione di produzione, così come quella di sviluppo, include anche la gestione di **MongoDB** come servizio persistente del cluster, caratterizzato da un’inizializzazione automatica del database attraverso lo script **init-mongo.js** che configura utenti e collections. Il sistema utilizza credenziali specifiche configurate tramite variabili d’ambiente e implementa una strategia di volume mounting che separa i dati persistenti (mongo:/data/db) dai certificati e configurazioni temporanee, garantendo che i dati importanti sopravvivano ai restart del cluster.

5.3.2.2 Deployment Automatizzato

Lo script **deploy-dccv.sh** rappresenta un esempio di automazione del deployment che implementa una sequenza completa di bootstrap del cluster distribuito (figura 15): lo script accetta un parametro di versione opzionale (se non specificato viene impostato a latest) che permette il deployment di release specifiche e la sua logica di inizializzazione include verifiche preliminari dell’installazione Docker, l’inizializzazione automatica di Docker Swarm con gestione dell’IP e il download automatico del file docker-compose dalla release GitHub specifica tramite l’URL costruito dinamicamente (<https://github.com/brunoesposito2/dccv>). Questa implementazione segue un metodo ’release-driven’ cioè in cui ogni versione del software include tutti gli artifacts necessari per il deployment, eliminando dipendenze dall’ambiente locale e garantendo riproducibilità assoluta. Il sistema utilizza anche tag Docker version-specific oltre a latest, permettendo rollback controllati e deployment di versioni specifiche in ambienti diversi.

Un’altra caratteristica dello script è la gestione automatizzata delle credenziali Swarm attraverso l’estrazione del join token (`docker swarm join-token -q worker`) e dell’IP del manager (`docker node inspect self --format '{{.Status.Addr}}'`), che vengono utilizzati per configurare dinamicamente le variabili d’ambiente necessarie per l’integrazione del cluster.

5.3.2.3 Monitoring e Resilienza

La topologia di rete implementata sfrutta le capacità native di service discovery di Docker Swarm, dove ogni servizio diventa automaticamente raggiungibile attraverso il suo nome DNS all’interno della rete overlay **swarm-network** (vedi figura 15). Questo elimina la necessità di configurazioni statiche di IP e permette ai componenti Akka di scoprirsì automaticamente attraverso i seed nodes configurati dinamicamente (cameranode:2553, guibackend:2551, utilitynode:2552), creando una topologia di cluster che si

adatta automaticamente all'aggiunta o rimozione di nodi senza richiedere riconfigurazioni manuali.

L'integrazione con **Portainer**, invece, richiede un deployment separato attraverso il file dedicato *portainer.yml* che espone l'interfaccia web sulla porta 9000 con volumi specifici per la persistenza della configurazione. Portainer si connette al Docker Swarm fornendo visibilità real-time sullo stato dei servizi, metriche di performance, gestione dei volumi e delle reti e capacità di scaling dinamico attraverso l'interfaccia utente. Gli amministratori infatti possono accedere al pannello di controllo tramite *http://localhost:9000* e gestire l'intero cluster distribuito (figura 15).

Inoltre, il sistema si affida anche al meccanismo di failure detection integrato in **Akka Cluster** per rilevare nodi non responsivi e gestire la riconfigurazione automatica del cluster, che rappresenta un approccio più specifico per sistemi ad attori distribuiti (come mostrato nella figura 15).

Dunque la **resilienza** del sistema deriva dalla combinazione di restart policies Docker, persistenza dello stato critico attraverso volumi e le capacità native di Akka Cluster per la gestione distribuita dei failure, creando un sistema che può sopravvivere a failure di componenti individuali.

5.4 Licenza

Il progetto è destinato all'utilizzo in contesti universitari (per progetti di ricerca e tesi) e aziendali, dove fornisce l'implementazione di canali di comunicazione e deployment di artefatti software complessi come programmi di visione artificiale. Per queste ragioni si è adottata la licenza LGPLv3 che, a differenza della GPLv3, permette l'integrazione in software proprietari senza imporre obblighi di licenza all'intero sistema, ampliando così le possibilità di adozione pur mantenendo questa specifica componente saldamente ancorata nell'ecosistema open source.

5.5 Continuous Integration: propagazione dei commit, versionamento, release e documentazione

Il sistema di Continuous Integration trasforma l'architettura layered del sistema in un workflow automatizzato di sviluppo, testing e deployment. L'implementazione consiste in **due workflow principali** che separano la gestione del flusso di sviluppo dalla gestione delle release, utilizzando due GitHub Actions specializzate.

L'approccio alla base di questa implementazione deriva direttamente dalla struttura Domain-Driven Design del progetto: mentre l'architettura del software è organizzata in layer che riflettono responsabilità funzionali diverse, il sistema CI/CD replica questa separazione creando pipeline automatizzate che rispettano e rafforzano i confini architettonici. Questo approccio garantisce che modifiche a un layer specifico vengano validate nel contesto appropriato prima di propagarsi attraverso i layer dipendenti, prevenendo la diffusione di errori e mantenendo l'integrità dell'architettura complessiva.

Una volta che le modifiche hanno attraversato con successo questa catena di validazione architettonica, interviene il secondo workflow che trasforma il codice validato in artefatti

distribuibili completando così il ciclo dalla modifica dello sviluppatore alla release pronta per il deployment.

5.5.1 Development Flow: Chain Merge Workflow

Il workflow di propagazione dei commit implementato in `.github/workflows/chainMerge.yml`, serve a mantenere la coerenza di sviluppo nell'architettura multi-layer del progetto. Questo approccio automatizza la propagazione delle modifiche attraverso una catena di branch che rispecchia la struttura del sistema: domain → application → presentation → storage → interface → distribution → main.

Il workflow viene attivato automaticamente su push ai branch architetturali (domain, application, presentation, storage, interface, distribution) escludendo modifiche ai file `settings.gradle.*` e `gradle.properties` (figura 16). La configurazione dell'ambiente include un servizio MongoDB containerizzato con credenziali specifiche e health checks automatici, garantendo che ogni layer che dipende dalla persistenza possa essere testato in un ambiente che replica tali condizioni. In particolare, il servizio viene configurato con utenti dedicati e database inizializzato, permettendo ai test di validare non solo la logica applicativa ma anche l'integrazione con il layer di persistenza.

Un aspetto importante del workflow è il trattamento differenziato del branch domain: poiché questo layer contiene componenti C++ che richiedono OpenCV e dipendenze native specifiche, il sistema implementa un testing containerizzato che costruisce un'immagine Docker personalizzata dal Dockerfile del domain layer e poi esegue i test all'interno di esso (vedi figura 16). In questo modo si è risolto il problema di come testare codice C++ con dipendenze native specifiche in un ambiente CI/CD cloud-based.

Per tutti gli altri branch, il workflow utilizza una strategia di testing più diretta attraverso `./gradlew ${subproject}:test`, dove il subproject viene determinato dinamicamente dal nome del branch corrente - sfruttando la corrispondenza diretta tra nomi dei branch e nomi dei moduli Gradle (figura 16).

Il meccanismo di propagazione automatica rappresenta un'altra caratteristica importante di questo workflow. In particolare, quando i test di un layer passano con successo, il sistema determina automaticamente il branch successivo dell'architettura utilizzando un array che mappa ogni layer al suo layer dipendente. Il sistema quindi esegue un merge automatico del branch corrente nel branch successivo, propagando le modifiche validate attraverso l'architettura del sistema.

Questa strategia di propagazione automatica implementa un aspetto relativo dell'architettura layered: le modifiche fluiscono dai layer più interni (domain) verso quelli più esterni (interface, distribution), garantendo che ogni layer riceva solo modifiche che sono già state validate nei contesti di tutti i layer sottostanti. Il risultato è un sistema che previene la diffusione di errori attraverso l'architettura e mantiene potenzialmente ogni layer in uno stato sempre deployabile.

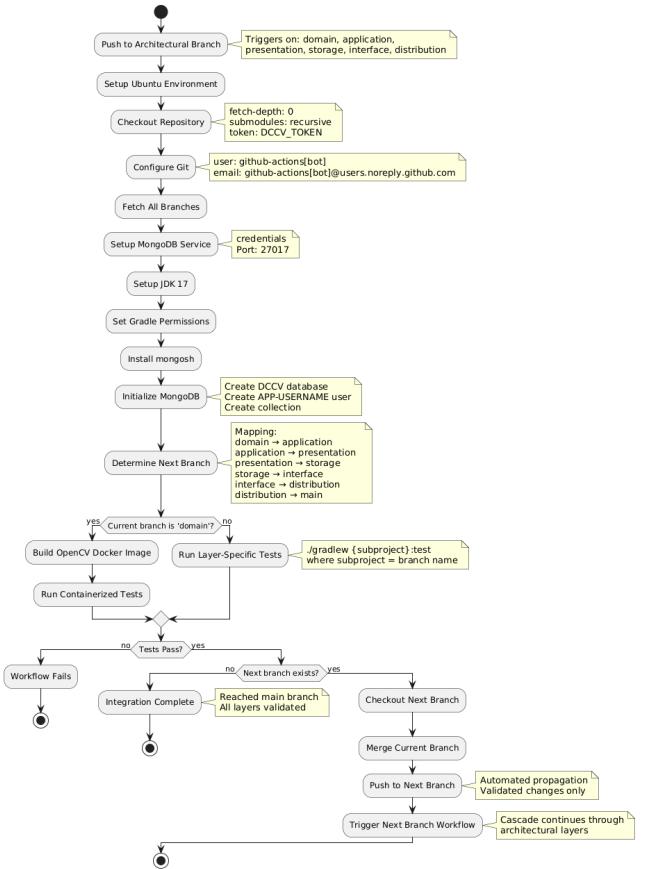


Figura 16: Activity diagram del Chain Merge Workflow

5.5.1.1 Analisi dei Tradeoff

Questa strategia di propagazione architettonale comporta alcuni trade-off che meritano una breve analisi critica. L'approccio risolve il problema di eventuali errori di integrazione nell'architettura multi-livello, dove modifiche apparentemente innocue possono creare effetti a cascata imprevedibili: la validazione progressiva garantisce che ogni layer riceva modifiche già testate nei layer più interni, riducendo così la probabilità di simili guasti. Tuttavia, la strategia può introdurre complessità operativa e potenziali bottleneck nel flusso di sviluppo poiché il processo sequenziale può rallentare il tempo di rilascio e la rigidità del flusso può risultare limitante per sviluppatori abituati a workflow più flessibili.

Nonostante queste potenziali limitazioni, l'implementazione del progetto considera questi aspetti attraverso metodi di mitigazione come il caching per ridurre i tempi di esecuzione e una giustificazione contestuale per sistemi di computer vision distribuiti dove errori di integrazione potrebbero compromettere la stabilità e l'affidabilità del cluster distribuito. Difatti è implementato un approccio fail-fast nel rilevamento degli errori e fail-safe nel

loro contenimento, privilegiando la robustezza rispetto alla velocità di delivery e risultando più appropriato per progetti di computer vision distribuita "orientati alla ricerca" - dove l'integrità architetturale può risultare più importante per validare i principi dimostrati - ma sovradimensionato per applicazioni che richiedono tempi di rilascio più rapidi.

5.5.2 Production Release Pipeline: Release Workflow

La pipeline di rilascio rappresenta l'evoluzione naturale del Chain Merge Workflow, in quanto trasforma le modifiche validate attraverso l'architettura layered in release complete e distribuite. Mentre il Chain Merge Workflow garantisce che le modifiche fluiscano correttamente attraverso i layer architetturali mantenendo l'integrità del sistema, il Release Workflow opera come un sistema di packaging e distribuzione che prende il risultato finale di questa validazione - il branch main - e lo trasforma in un insieme di artifacts deployabili.

Quando una modifica raggiunge il branch main attraverso la catena di validazione implementata dal chain merge workflow, essa rappresenta non solo codice funzionalmente corretto, ma anche un cambiamento che rispetta tutti i confini e le dipendenze dell'architettura. Il Release Workflow riconosce questo stato raggiunto e implementa l'automazione che lo trasforma in un rilascio del software concreto.

Il sistema implementato in `.github/workflows/release-workflow.yml` utilizza una logica condizionale per riconoscere automaticamente quando le modifiche al branch main costituiscono effettivamente una release significativa del sistema in base all'analisi semantica dei messaggi di commit. In questo modo si elimina la necessità di decisioni manuali sui tempi di release, permettendo di concentrarsi sulla logica applicativa mentre l'automazione gestisce la complessità della distribuzione.

Il sistema coordina simultaneamente operazioni parallele di build delle immagini Docker, generazione della documentazione e preparazione degli scripts di deployment, per poi convergere in una fase finale di sincronizzazione e pubblicazione tra tutti i componenti della release, assicurando che ogni release includa tutti gli elementi necessari per un deployment autonomo e riproducibile (figura 17).

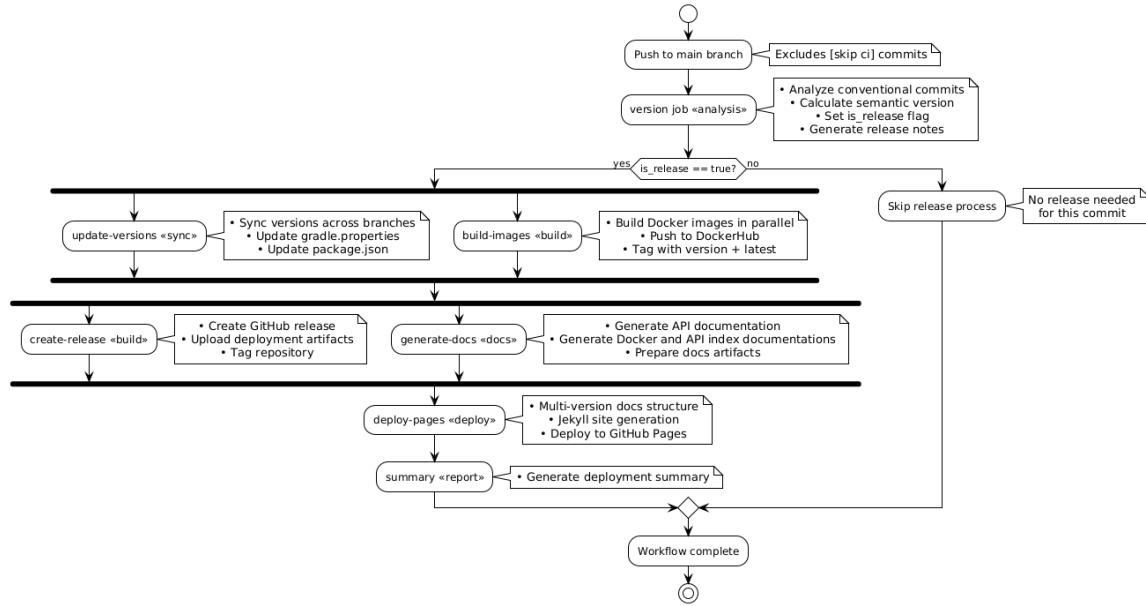


Figura 17: Activity diagram del Release Workflow

5.5.2.1 Versionamento Semantico e Trigger Logic

Il workflow di release implementato in `.github/workflows/release-workflow.yml`, rappresenta un sistema completo di versionamento semantico automatizzato che trasforma il branch main in un hub di release. Questo workflow viene attivato esclusivamente su push al branch main - quando il commit non contiene la stringa "skip ci" - e implementa una strategia di versionamento basata sull'analisi automatica dei commit messages seguendo le convenzioni dei Conventional Commits (job **version** nell'immagine 17).

In particolare, il sistema di determinazione della versione analizza lo storico dei commit messages dal tag precedente utilizzando pattern regex per categorizzare i cambiamenti. Il workflow riconosce breaking changes attraverso pattern come **feat!:** e **fix!:** o tramite la presenza esplicita di **BREAKING CHANGE** nel messaggio, l'aggiunta di feature attraverso i pattern **feat:** o **feat(scope):** e correzioni di bug tramite **fix:** o **fix(scope):**. Questa analisi determina automaticamente se incrementare la versione MAJOR (breaking changes), MINOR (nuove feature) o PATCH (bug fixes), implementando così la specifica di Semantic Versioning.

Il processo di creazione delle release notes, invece, categorizza automaticamente i commit in sezioni dedicate (Breaking Changes, New Features, Bug Fixes) e genera descrizioni strutturate che includono link rapidi per il deployment, informazioni sui servizi del sistema e istruzioni per l'utilizzo. Le release notes, inoltre, integrano automaticamente i link alle immagini Docker pubblicate, i comandi di deployment rapido e una tabella dei servizi con porte e scopi specifici.

Parallelamente, tramite il job dedicato (**summary**, vedi figura 17), il workflow genera un report riassuntivo automatizzato che trasforma l'output del processo CI/CD in un'interfaccia utente. Al termine di ogni release, il sistema genera automaticamente un report strutturato che include link diretti alla release GitHub appena creata, comandi di deployment pronti per il copy-paste e tabelle complete dei servizi con porte e finalità specifiche. Questo report viene integrato direttamente nell'interfaccia GitHub Actions, permettendo un accesso più rapido a tutte le informazioni necessarie per il deployment. Un aspetto fondamentale del workflow è la generazione automatica di artifacts di deployment (job **create-release** nell'immagine 17) che trasforma ogni release in un pacchetto autosufficiente. Il sistema genera automaticamente e allega alla release GitHub diversi componenti: lo script `deploy-dccv.sh` che contiene la logica completa di bootstrap del cluster, il file `docker-compose-prod.yml` configurato per la versione specifica e una struttura completa di documentazione. In questo modo ogni versione include tutti gli elementi necessari per un deployment autonomo eliminando le dipendenze dall'ambiente di sviluppo. Un utente può quindi scaricare una release specifica e avere immediatamente accesso a tutti gli strumenti necessari per deployare quella versione esatta del sistema, facilitando così i rollback controllati e deployment riproducibili in ambienti diversi. Il workflow implementa inoltre un sistema di orchestrazione condizionale che ottimizza l'utilizzo delle risorse CI/CD attraverso dipendenze tra job ed esecuzione condizionale. Il sistema utilizza la direttiva **needs** per creare un grafo di dipendenze esplicito dove, ad esempio, il job **build-images** dipende dal completamento di **version** ed esegue solo quando `needs.version.outputs.is_release == 'true'`. Questo garantisce che operazioni come il build e la pubblicazione su DockerHub delle immagini Docker avvengano esclusivamente quando necessario, evitando sprechi di risorse computazionali per commit che non generano release. Analogamente, il job **deploy-pages** (in figura 17) dipende sia da **version** che da **generate-docs**, creando una catena di dipendenze che assicura che la documentazione venga deployata solo dopo che tutti i prerequisiti siano stati soddisfatti con successo.

5.5.2.2 Operazioni di Build Parallela

Il workflow include il job **build-images** (figura 17) che automatizza la creazione e pubblicazione delle immagini Docker su DockerHub. Il sistema utilizza una matrice per parallelizzare la build di tutte le immagini (cameranode, guifrontend, guibackend, utilitynode), ognuna configurata con context e Dockerfile specifici, e la loro successiva pubblicazione su DockerHub. Ogni immagine viene taggata sia con la versione specifica della release che con il tag latest, permettendo deployment sia version-locked che rolling updates. Il sistema bilancia parallelismo e dipendenze con i job poiché mentre i build Docker avvengono in parallelo mantengono la dipendenza dal job **version** affinché utilizzino i tag corretti.

La configurazione include strategie di caching come `cache-from: type=registry` e `cache-to: type=registry` che ottimizzano i tempi di build riutilizzando layer precedentemente costruiti - questo approccio riduce i tempi di CI/CD per modifiche incrementali.

5.5.2.3 Generazione Documentazione e Deployment

Il sistema implementa una gestione degli artifacts che permette la condivisione di file e dati tra job diversi attraverso un meccanismo di upload/download automatizzato. Questo sistema è utile per l'approccio multi-job del workflow, dove il job **generate-docs** produce documentazione API che deve essere successivamente utilizzata dal job **deploy-pages** (figura 17). Il meccanismo utilizza **actions/upload-artifact@v4** per salvare la documentazione generata come artifact nominato api-docs, che viene poi scaricato tramite **actions/download-artifact@v4** nel job di deployment. Questa strategia permette ad ogni job di concentrarsi su responsabilità specifiche mantenendo la capacità di condividere output con job dipendenti.

Il workflow completo di generazione della documentazione crea automaticamente una documentazione API strutturata per ogni release. Il processo estrae informazioni principali dai file docker-compose - come nomi dei servizi, porte utilizzate e altro -, genera documentazione sia per i servizi Scala utilizzando Gradle e sia per gli endpoint REST utilizzando la specifica openapi (definita in *interface/server/src/main/resources*), e crea strutture di documentazione versionate che permettono di mantenere una documentazione storica per ogni release.

Il deployment avviene attraverso GitHub Pages utilizzando **Jekyll** per la generazione del sito statico. Il sistema configura automaticamente domini personalizzati, certificati SSL e strutture di navigazione che permettono di accedere facilmente sia alla documentazione corrente che a versioni precedenti. La documentazione include sezioni dedicate per **API Scala** e **REST** e **configurazioni Docker**, creando delle risorse complete relative all'**API pubblica** del progetto consultabile via web.

5.5.2.4 Sincronizzazione Cross-Branch

Un altro aspetto del workflow è il sistema di sincronizzazione delle versioni attraverso tutti i branch architetturali. Dopo la creazione di una release, il sistema aggiorna automaticamente i file gradle.properties e package.json in tutti i branch garantendo che ogni layer mantenga la versione corretta e che sviluppi futuri partano sempre dalla versione corretta.

La combinazione di tutti questi elementi crea un sistema di Continuous Integration che trasforma ogni modifica del codice in una sequenza automatizzata di validazione, testing, propagazione, versionamento e deployment, in modo tale che il sistema rimanga sempre in uno stato "production-ready" supportando al tempo stesso la velocità di iterazione per il development.

6 Istruzioni per il deployment

Questa sezione descrive le procedure per il deployment dell'applicazione DCCV, distinguendo tra l'ambiente di sviluppo e l'ambiente di produzione.

6.1 Deployment in Ambiente di Produzione

L'ambiente di produzione è progettato per essere distribuito e scalabile, utilizzando Docker Swarm per l'orchestrazione dei container.

6.1.1 Deployment Automatizzato

Il metodo più semplice per il deployment è utilizzare lo script **deploy-dccv.sh**. Per farlo è possibile seguire il seguente procedimento:

1. **Scaricare lo script di deploy:** Ottenere lo script deploy-dccv.sh dalla release ufficiale del progetto. Lo script può accettare un parametro opzionale per specificare la versione da deployare; se omesso, verrà utilizzata la versione latest;
2. **Rendere lo script eseguibile:**

```
1      chmod +x deploy-dccv.sh
```

3. **Avviare il deployment:**

```
1      ./deploy-dccv.sh [VERSIONE]
```

Lo script eseguirà le seguenti operazioni:

- **Verifica dei prerequisiti:** Controlla la presenza di Docker e la sua configurazione;
- **Inizializzazione Docker Swarm:** Configura automaticamente il cluster Swarm se non presente;
- **Configurazione di rete:** Crea la rete overlay swarm-network per la comunicazione inter-container;
- **Download delle configurazioni:** Recupera automaticamente il file docker-compose-prod.yml dalla release GitHub;
- **Deployment dello stack:** Avvia tutti i servizi utilizzando Docker Stack.

Il sistema supporta il versionamento semantico automatico, permettendo il deployment di versioni specifiche o dell'ultima versione disponibile.

6.1.2 Deployment Manuale

In alternativa, è possibile eseguire il deployment manualmente in questo modo:

1. **Inizializzare Docker Swarm:**

```
1 docker swarm init
```

2. **Creare la Rete Overlay:**

```
1 docker network create --driver overlay --attachable swarm  
-network
```

3. **Eseguire il Deploy dello Stack** (assicurarsi prima di aver scaricato il file **docker-compose-prod.yml** dalla release Github del progetto):

```
1 docker stack deploy -c docker-compose-prod.yml dccv
```

6.1.3 Architettura di Deployment

Il deployment è organizzato attraverso cinque servizi containerizzati principali che verranno attivati indipendentemente dall'ambiente:

- **CameraNode:** usa l'immagine docker brunoesposito2/dccv-cameranode e gestisce l'elaborazione computer vision utilizzando le porte 2553, 5555, 8080;
- **GUI Frontend:** usa l'immagine docker brunoesposito2/dccv-guifrontend e rappresenta l'interfaccia React disponibile sulla porta 3000;
- **GUI Backend:** usa l'immagine brunoesposito2/dccv-guibackend, serve per la parte di API REST e di coordinamento Akka e lavora sulle porte 2551 e 4000;
- **UtilityNode:** usa l'immagine brunoesposito2/dccv-utilitynode, permette la supervisione del cluster ed usa la porta 2552;
- **MongoDB:** persistenza dati su porta 27017.

Una volta completato il deployment, l'interfaccia web del sistema sarà accessibile all'indirizzo <http://localhost:3000>.

6.2 Deployment in Ambiente di Sviluppo

L'ambiente di sviluppo è ottimizzato per la produttività degli sviluppatori fornendo strumenti per il testing e il debugging.

6.2.1 Deployment Automatizzato per Sviluppo

Il sistema fornisce lo script `dev-utils.sh` che automatizza la gestione dell'ambiente di sviluppo. Per utilizzarlo è necessario seguire il seguente procedimento:

1. **Navigare nella directory distribution:**

```
1      cd distribution
```

2. **Rendere lo script eseguibile:**

```
1      chmod +x dev-utils.sh
```

3. **Avviare l'ambiente di sviluppo:**

```
1      ./dev-utils.sh start-dev
```

Lo script eseguirà automaticamente le seguenti operazioni:

- **Pulizia dei lock file Gradle:** Rimuove eventuali file di lock che potrebbero impedire la compilazione;
- **Avvio sequenziale dei nodi:** Avvia CameraNode, UserNode e UtilityNode in modalità sviluppo;
- **Configurazione hot-reload:** Abilita il ricaricamento automatico del codice modificato.

6.2.2 Sincronizzazione Incrementale

Per poter sincronizzare selettivamente i singoli servizi senza dover riavviare l'intero stack è possibile utilizzare i seguenti comandi dello script:

- **Sincronizzazione CameraNode:**

```
1      ./dev-utils.sh sync-camera
```

- **Sincronizzazione UserNode (Backend):**

```
1      ./dev-utils.sh sync-user
```

- **Sincronizzazione UtilityNode:**

```
1      ./dev-utils.sh sync-utility
```

- **Sincronizzazione completa:**

```
1      ./dev-utils.sh sync-all
```

6.2.3 Gestione del Ciclo di Vita di Sviluppo

L'ambiente di sviluppo fornisce anche comandi per la gestione del ciclo di vita:

- **Arresto dei Servizi**

```
1      ./dev-utils.sh stop
```

- **Pulizia Completa**, per rimuovere completamente l'ambiente e pulire tutte le risorse:

```
1      ./dev-utils.sh clean
```

Una volta completato il deployment di sviluppo, i servizi saranno accessibili agli stessi indirizzi dell'ambiente di produzione, facilitando il testing e la transizione tra ambienti.

7 Conclusioni

7.1 Risultati Raggiunti e Conclusioni

Il progetto DCCV ha raggiunto l'obiettivo di creare un framework distribuito per applicazioni di computer vision che combina sia l'efficienza computazionale di C++ con la robustezza e scalabilità di un'architettura ad attori implementata in Scala, e sia i principi e le pratiche dell'ingegneria dei processi software con lo sviluppo di sistemi distribuiti. L'implementazione ha infatti permesso di integrare tecnologie diverse attraverso un'architettura layered, mantenendo al tempo stesso la separazione delle responsabilità e la manutenibilità del codice.

Dal punto di vista architettonico, l'adozione del Domain Driven Design si è rivelata efficace nel gestire la complessità del sistema: la separazione in layer distinti ha permesso di evolvere indipendentemente le diverse componenti tecnologiche, facilitando sia lo sviluppo parallelo da parte del team, sia l'introduzione di modifiche senza impatti a cascata sull'intero sistema. Inoltre, il modello ad attori implementato con Akka ha permesso la gestione della concorrenza e della distribuzione permettendo al sistema di scalare orizzontalmente con l'aggiunta di nuovi nodi.

L'automazione DevOps è risultata particolarmente importante durante tutto il ciclo di sviluppo: la pipeline di CI implementata con GitHub Actions ha prevenuto la propagazione di errori attraverso l'architettura multi-layer, mentre il sistema di versionamento semantico automatizzato ha semplificato il processo di release. In più, la containerizzazione completa del sistema ha eliminato i problemi di configurazione dell'ambiente, rendendo il deployment riproducibile e predicable.

Tra le diverse sfide tecniche affrontate, l'integrazione tra il layer C++ per la computer vision e il sistema Scala ad attori ha richiesto una particolare attenzione nella gestione dei confini tecnologici. La soluzione adottata, basata su comunicazione via socket e streaming reattivo con Akka Streams, ci ha permesso di gestire efficacemente il throughput richiesto per lo streaming video in tempo reale, mantenendo latenze accettabili per applicazioni di monitoraggio.

7.2 Sviluppi futuri

L'architettura modulare implementata fornisce le basi per diverse direzioni di evoluzione del sistema. Nel breve termine, l'integrazione di algoritmi di computer vision più sofisticati rappresenta il naturale passo successivo tramite ad esempio l'implementazione di tecniche per il riconoscimento facciale, il tracking multi-oggetto ed altro: l'architettura esistente supporta già questa evoluzione, richiedendo modifiche limitate al solo layer domain. Dal punto di vista dell'orchestrazione, la migrazione verso Kubernetes potrebbe portare a maggiori benefici in termini di gestione delle risorse, auto-scaling basato su metriche e gestione dichiarativa delle configurazioni.

Infine, il miglioramento dell'interfaccia utente verso una dashboard più ricca con capacità di analytics in tempo reale e sistemi di alerting configurabili, aumenterebbe il valore applicativo del sistema.

In generale, questo percorso di sviluppo del progetto ci ha permesso di produrre un sistema articolato ma manutenibile dove l'investimento iniziale nell'architettura e nell'automazione si è tradotto in benefici a lungo termine in termini di evolvibilità e affidabilità del software prodotto.