

Virtualizzazione e Integrazione di sistemi Podman e Industria 4.0

Gambacorta Giuseppe

4 settembre 2025

1 Introduzione

Lo scopo di questo progetto è approfondire lo studio delle tecnologie di containerizzazione, con particolare attenzione a Podman, analizzandone le caratteristiche e confrontandolo con Docker, la piattaforma più diffusa in questo ambito.

Una volta comprese le differenze tra i due strumenti, l'obiettivo pratico del progetto sarà l'implementazione di un pod basato su Podman orientato all'Industria 4.0. In particolare, si realizzerà un sistema capace di raccogliere e visualizzare dati provenienti da dispositivi o processi industriali, con l'intento di mostrare come l'approccio a container possa semplificare lo sviluppo e la distribuzione di architetture modulari e facilmente scalabili.

È compresa anche un'introduzione ai PLC (Programmable Logic Controller), descrivendo il loro ruolo nell'automazione industriale, e una panoramica sui principali protocolli di comunicazione industriale (come Modbus, OPC UA e Ethercat), che costituiscono la base per l'integrazione tra macchine, sistemi e applicazioni informatiche. Questa parte introduttiva permette di comprendere meglio il contesto applicativo in cui le tecnologie di containerizzazione possono trovare impiego.

2 OCI

In ambito container, **OCI** (Open Container Initiative) rappresenta uno standard aperto e condiviso che definisce come devono essere costruiti e gestiti i container e le loro immagini. L'obiettivo principale è assicurare che i container possano essere eseguiti in maniera coerente e compatibile su diversi runtime, come Docker, Podman e altri, senza richiedere conversioni o adattamenti complessi. Più nello specifico:

- **OCI Image Format** Definisce come deve essere strutturata un'immagine di container (file system, metadati, layer, manifest, ecc.). Grazie a questo standard, un'immagine creata con Docker può essere eseguita con Podman, rkt o altri runtime compatibili. In pratica, l'immagine contiene tutto ciò che serve al container per partire: sistema operativo minimale, applicazioni, librerie e configurazioni.
- **OCI Runtime Specification** Definisce come deve essere eseguito un container, ovvero come il runtime deve avviare e isolare il processo containerizzato. Include dettagli su:
 - **Processi:** come il container viene avviato e quali processi vengono eseguiti all'interno.
 - **Filesystem:** come i layer dell'immagine vengono montati e isolati dal resto del sistema host.
 - **Rete:** configurazione di interfacce di rete virtuali e isolamento dai processi host.
 - **Cgroups e risorse:** gestione di CPU, memoria, I/O e altre risorse per garantire limiti e isolamento.

In altre parole, il runtime è il “motore” che trasforma un'immagine OCI in un container funzionante e isolato, rispettando le regole definite dallo standard.

Per ulteriori informazioni: [Open Container Initiative](#)

3 Podman

Podman è nato dopo Docker e ha avuto il vantaggio di poter osservare il design di Docker con una prospettiva completamente nuova. I sviluppatori di Podman hanno studiato le criticità di Docker e hanno cercato di migliorarne l'architettura, mantenendo la compatibilità grazie al codice open source condiviso. Inoltre, Podman ha potuto sfruttare nuovi standard emergenti, come quelli definiti dalla Open Container Initiative (OCI), per garantire portabilità e interoperabilità tra diversi strumenti di containerizzazione.

3.1 Principali differenze con Docker

Sebbene Docker sia stato il pioniere nella containerizzazione, Podman è emerso come un'alternativa moderna e innovativa, progettata per affrontare alcune delle limitazioni di Docker. Una delle differenze fondamentali risiede nell'architettura: Docker si basa su un modello client-server, con un demone centrale (dockerd) che gestisce i container. In contrasto, Podman adotta un'architettura senza demone, dove ogni comando è eseguito come un processo indipendente, migliorando la sicurezza e riducendo la superficie di attacco.

Un aspetto distintivo di Podman è la gestione dei pod. I pod in Podman sono gruppi di container che condividono lo stesso spazio di rete e possono essere gestiti collettivamente. Questo concetto, mutuato da Kubernetes, consente una gestione più efficiente e coerente dei container correlati, facilitando la transizione verso ambienti di orchestrazione come Kubernetes.

In termini di compatibilità, Podman offre una CLI simile a quella di Docker, permettendo agli utenti di utilizzare comandi familiari. Inoltre, strumenti come podman-compose offrono funzionalità analoghe a Docker Compose, consentendo la definizione e l'esecuzione di applicazioni multi-container.

Dal punto di vista della sicurezza, Podman supporta l'esecuzione di container senza privilegi di root (rootless), riducendo significativamente i rischi associati alle escalation di privilegi.

3.2 Daemonless

Docker è costruito come un server basato su REST API e adotta un'architettura client-server che include più demoni. Quando un utente esegue il client Docker, questo si connette al demone Docker, che gestisce il download delle immagini e l'esecuzione dei container.

Il demone Docker gira come root e funge da piattaforma centrale per la gestione dei container. Gli utenti possono avviare applicazioni containerizzate senza rendersi conto della complessità che avviene dietro le quinte.

In sintesi, l'esecuzione dei container con Docker richiede più demoni che devono rimanere attivi continuamente, e guasti in uno di essi possono causare l'arresto di tutti i container, rendendo difficile diagnosticare eventuali problemi.

Podman è fondamentalmente diverso da Docker perché non utilizza un demone. Podman può eseguire tutte le stesse immagini di container di Docker e avviare container con gli stessi runtime, ma senza avere più demoni in esecuzione continua come root. I container continuano a funzionare senza il sovraccarico derivante dall'esecuzione di più demoni, un vantaggio particolarmente apprezzato su macchine poco potenti, come dispositivi IoT.

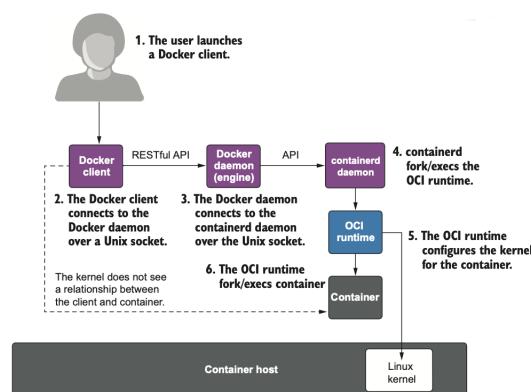


Figura 1: Architettura Docker

3.2.1 Fork/Exec model

In Linux, l'avvio di un nuovo programma avviene tramite due operazioni distinte: fork ed exec. La chiamata di sistema fork genera un nuovo processo figlio, identico al processo padre, duplicandone lo stato iniziale. Successivamente, il processo figlio può invocare exec per sostituire il proprio spazio di esecuzione con un nuovo programma. In tal modo, il processo mantiene il suo identificatore nel sistema operativo, ma esegue un codice completamente differente.

Podman adotta lo stesso principio. Quando viene avviato un container, il sistema crea un processo figlio tramite fork e successivamente lo trasforma nel container mediante exec, eseguendo il runtime OCI appropriato. Il processo Podman originario termina subito dopo, mentre il container continua a operare come processo indipendente del sistema operativo. Questo modello daemonless elimina la necessità di un demone sempre attivo, riduce l'overhead complessivo e consente ai container di proseguire la loro esecuzione anche durante aggiornamenti o modifiche al motore di container.

Il funzionamento di Podman si basa quindi sul modello Fork/Exec: all'avvio di un container o quando ci si riconnette a un container già in esecuzione, il processo Podman temporaneo termina subito dopo aver creato o collegato il container, lasciandolo funzionare come processo indipendente del sistema operativo.

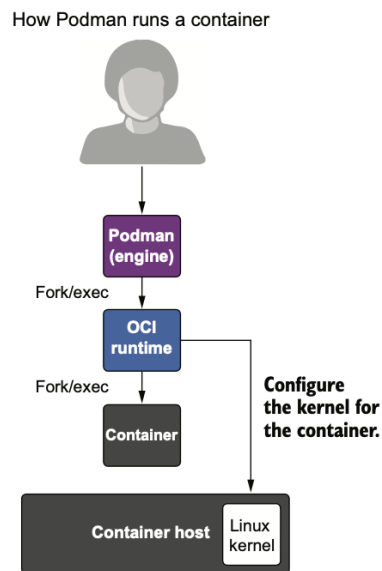


Figure 1.9 Podman fork/exec architecture. The user launches Podman, which executes the OCI runtime, which then launches the container. The container is a direct descendant of Podman.

Figura 2: Architettura Podman

Un altro vantaggio di questo approccio è che i container continuano a funzionare anche durante aggiornamenti del sistema o modifiche al software del motore di container, poiché non esiste un demone centrale che debba essere riavviato durante un update. Le applicazioni containerizzate non subiscono interruzioni, garantendo maggiore continuità di servizio.

3.2.2 RestAPI

Nonostante Podman sia daemonless, può essere eseguito come servizio socket-activated che espone una REST API. Questo permette a client remoti di gestire e lanciare container Podman in modo simile a Docker. Grazie al supporto sia per l'API Docker sia per l'API nativa di Podman, è possibile continuare a utilizzare strumenti e script già sviluppati per Docker senza modifiche significative, pur beneficiando delle funzionalità avanzate offerte da Podman.

Quando il servizio è attivo, un socket systemd resta in ascolto delle richieste dei client. Questo significa che non esiste un daemon Podman sempre in esecuzione: il socket, gestito da systemd, avvia automaticamente Podman solo quando arriva una richiesta. In pratica, systemd fa da "intermediario", attivando il processo Podman al bisogno e chiudendolo quando non è più necessario.

3.3 Rootless

Linux è stato progettato fin dall'inizio con una separazione tra modalità privilegiata (rootful) e modalità non privilegiata (rootless). Su Linux, quasi tutti i processi vengono eseguiti senza privilegi di root. Operazioni privilegiate sono necessarie solo per modifiche al sistema operativo di base. La maggior parte delle applicazioni che girano nei container — come web server, database e strumenti per l'utente — non richiedono privilegi di root, perché non modificano parti fondamentali del sistema.

Purtroppo, la maggior parte delle immagini presenti nei registry di container sono costruite per richiedere privilegi di root, o almeno partono come root e poi riducono i privilegi.

Quando si utilizza un container tramite un daemon rootful, come il Docker daemon avviato come root, l'utente root all'interno del container corrisponde effettivamente al root del sistema host. Questo significa che i processi all'interno del container possono avere privilegi elevati e, se configurati con accesso a dispositivi o directory dell'host, potrebbero modificare parti del sistema.

Al contrario, con container rootless, come quelli eseguiti da Podman o Docker in modalità rootless (nelle nuove versioni), l'utente root all'interno del container non corrisponde a root sul l'host, ma a un UID non privilegiato. In questo modo è possibile eseguire container senza rischi di compromettere il sistema host, mantenendo l'isolamento e la sicurezza tipici dei container.

3.3.1 User-Namespaces

Podman è completamente integrato con i user namespaces. La modalità rootless si basa sui user namespace, che permettono di assegnare più UID (User ID) a un singolo utente. In pratica, un user namespace consente di isolare i container tra loro e dall'host, così più utenti rootless possono eseguire container separati, ognuno con propri UID, senza interferire l'uno con l'altro.

- **UID (User ID):** è un numero che identifica un utente sul sistema Linux. L'UID 0 corrisponde a **root**.
- **GID (Group ID):** è un numero che identifica un gruppo di utenti.

Nei container rootless, l'utente **root** all'interno del container non corrisponde all'UID 0 dell'host, ma a un UID non privilegiato mappato dall'host. Questo permette di eseguire container come **root** senza concedere realmente privilegi di root sul sistema host.

Podman rende semplice l'esecuzione di più container, ciascuno con un user namespace unico. Il kernel isola quindi i processi non solo dagli utenti dell'host, ma anche dagli altri container, basandosi sulla separazione degli UID.

Al contrario, Docker supporta solo un singolo user namespace separato per tutti i container: tutti i container condividono lo stesso mapping UID → UID host. Questo significa che root in un container è uguale a root in un altro container, e non c'è isolamento completo dal punto di vista del user namespace. Sebbene Docker supporti modalità con user namespace separati, nella pratica quasi nessuno la utilizza.

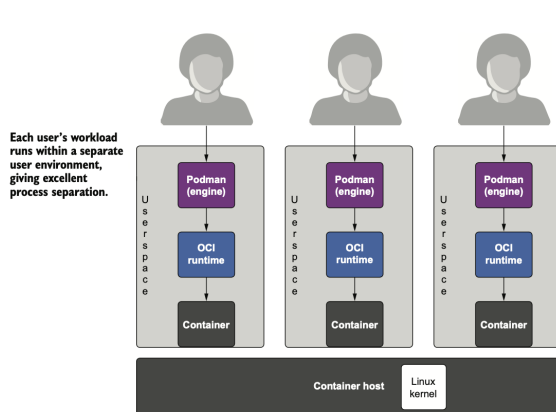


Figura 3: Podman rootless

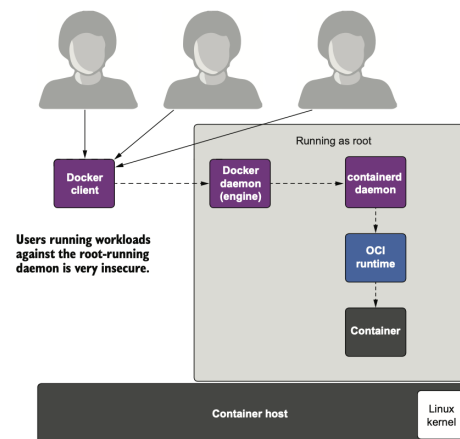


Figura 4: Docker rootful

3.4 CLI

Uno dei grandi punti di forza di Docker è la sua interfaccia a riga di comando semplice ed intuitiva. Esistono altri strumenti per container come RKT, LXC e LXD, ma ognuno ha la propria CLI. Il team di Podman ha presto capito che non avrebbe conquistato quote di mercato se Podman avesse avuto un'interfaccia completamente diversa. Docker era lo strumento dominante e quasi tutti coloro che avevano già usato i container lo avevano fatto tramite la sua CLI.

Inoltre, se si cercava online come fare qualcosa con un container, quasi sempre si trovava un esempio usando la riga di comando di Docker. Fin dall'inizio, quindi, Podman doveva essere compatibile con la CLI di Docker. È nato rapidamente uno slogan per sostituire Docker con Podman: `alias docker=podman`

Con questo comando, è possibile continuare a digitare i comandi Docker, ma a eseguirli sarà Podman. Se la CLI di Podman differisce da quella di Docker, viene considerato un bug e gli utenti richiedono che venga corretto per rendere i due strumenti equivalenti. Esistono alcune funzionalità che Podman non supporta, come Docker Swarm (per l'orchestrazione, Podman è pensato per integrarsi con Kubernetes), ma per la maggior parte dei comandi Podman rappresenta un sostituto completo della CLI Docker.

3.5 Systemd e l'integrazione con Podman

Systemd è il sistema di init fondamentale per molti sistemi operativi Linux. L'**init** è il primo processo avviato dal kernel al boot del sistema e rappresenta l'antenato di tutti gli altri processi. Di conseguenza, systemd ha la capacità di monitorare, gestire e coordinare tutti i processi del sistema, compresi quelli dei container.

Podman è progettato per integrarsi pienamente con systemd, permettendo di gestire i container come **servizi di sistema**, sfruttando tutte le funzionalità avanzate di systemd. Tra le principali caratteristiche supportate:

- Supporto a systemd **all'interno del container**, permettendo di eseguire più servizi definiti dal sistema in un singolo container.
- Supporto alla **socket activation**, cioè l'avvio di processi su richiesta quando un socket riceve una connessione.
- Notifiche a systemd che indicano quando un'applicazione containerizzata è completamente avviata.
- Gestione completa di **cgroups** e del ciclo di vita delle applicazioni containerizzate.

In pratica, con Podman un container può funzionare come un servizio systemd: systemd diventa responsabile del suo avvio, spegnimento e monitoraggio, come avverrebbe per qualsiasi altro servizio del sistema. Quando si avvia systemd all'interno di un container, Podman configura l'ambiente in modo che systemd possa eseguire i processi come **PID 1** del container, garantendo compatibilità completa.

Al contrario, Docker non supporta l'integrazione diretta con systemd. La comunità Docker ritiene che sia il **daemon Docker** a dover gestire il ciclo di vita dei container, senza delegare funzioni a systemd. Questo limita alcune funzionalità avanzate come l'ordinamento dell'avvio, socket activation o le notifiche di servizio pronto, che invece Podman supporta.

3.5.1 Systemd interno

Systemd è il sistema di init di Linux, responsabile dell'avvio e della gestione dei processi e dei servizi. All'interno di un container Podman, è possibile eseguire systemd come **PID 1**, cioè il processo principale del container. In questo modo:

- Il container può avviare più servizi interni come in un sistema Linux completo.
- Systemd gestisce il ciclo di vita dei processi e dei servizi all'interno del container.
- In modalità rootless, systemd funziona senza privilegi di root sull'host, usando UID mappati tramite user namespaces.

Vantaggio principale: permette di trattare i container come veri e propri sistemi Linux miniaturizzati, con gestione completa dei servizi.

3.5.2 Socket Activation

La **socket activation** è una tecnica che permette di avviare un servizio solo quando arriva una connessione su un socket specifico. In pratica:

- Systemd dentro il container ascolta su un socket (ad esempio una porta TCP).
- Il servizio collegato al socket viene avviato solo al momento della richiesta, evitando di mantenere processi inattivi.

Vantaggi:

- Migliora l'efficienza delle risorse, avviando servizi solo quando necessari.
- Migliora la sicurezza, riducendo l'esposizione dei servizi.
- Si integra perfettamente con systemd, sfruttando tutte le sue funzionalità di notifica e orchestrazione.

3.5.3 Cgroups (Control Groups)

I **cgroups** sono una funzionalità del kernel Linux che permette di:

- Limitare e isolare l'uso delle risorse (CPU, memoria, I/O) di un gruppo di processi.
- Contabilizzare l'utilizzo delle risorse per monitoraggio e controllo.

All'interno dei container Podman:

- Systemd può gestire i cgroups dei processi interni, anche in modalità rootless.
- Ogni servizio o processo può avere limiti separati, evitando che un container o servizio consumi tutte le risorse.
- Garantisce isolamento tra container e rispetto delle risorse dell'host.

Vantaggio principale: combina sicurezza, isolamento e gestione efficiente delle risorse, come in un sistema Linux completo, ma in ambiente containerizzato.

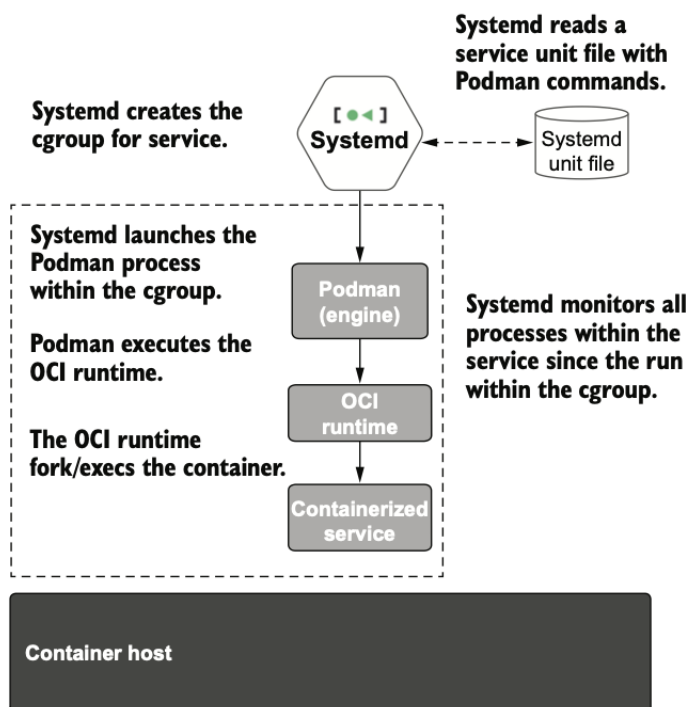


Figure 7.1 Systemd executing a Podman container

Figura 5: Podman tramite Systemd

3.6 Pods

Uno dei vantaggi principali di Podman è già suggerito dal suo nome: **Podman** è l'abbreviazione di Pod Manager. Uno degli obiettivi principali dei container è separare i servizi in container singoli, secondo il paradigma delle microservizi. Successivamente, i container possono essere combinati per costruire servizi più complessi. I pod permettono di raggruppare più container insieme, gestendoli come un'unica entità. Questo facilita esperimenti e test di architetture più complesse, mantenendo però il controllo su ogni singolo container.

Podman include il comando `podman generate kube`, che permette di generare file YAML compatibili con Kubernetes a partire da container e pod in esecuzione. Questo rende semplice il passaggio dallo sviluppo locale alla distribuzione su un cluster Kubernetes

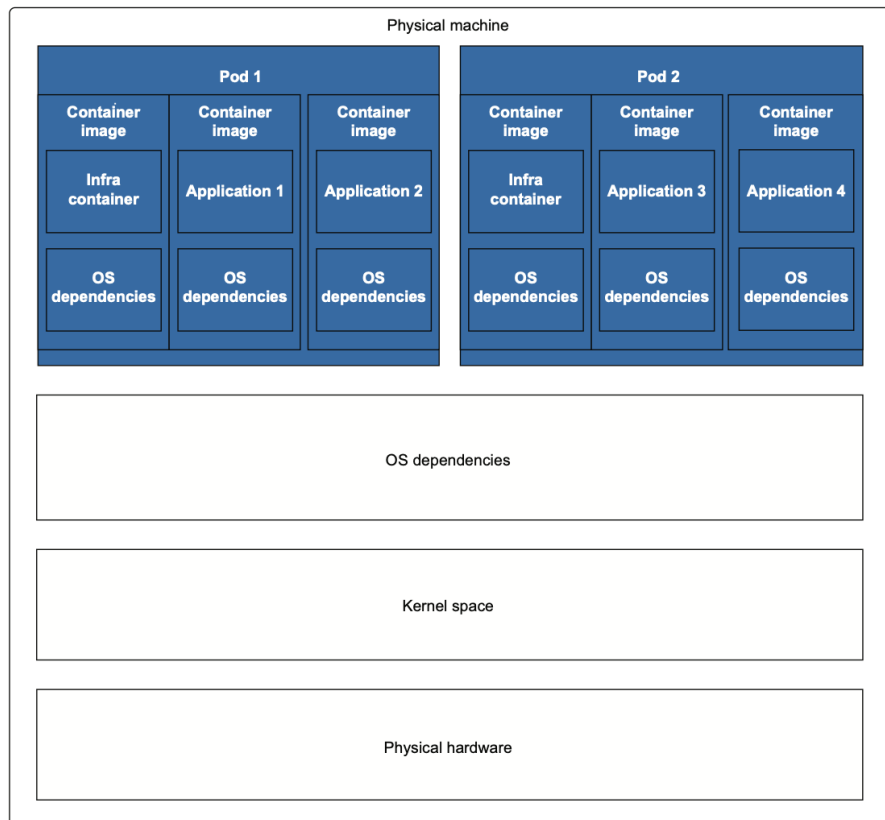


Figure 1.12 Two pods running on a host. Each pod runs two different containers along with the infra container.

Figura 6: Pods

3.6.1 Comunicazione interna nei pod

Tutti i container di un pod condividono lo stesso **network namespace**, quindi possono comunicare tra loro tramite `localhost`. Questo significa che un servizio in un container può raggiungere un altro container dello stesso pod semplicemente usando `127.0.0.1` e la porta esposta. Lo stesso principio vale per la condivisione di volumi e altri namespace.

3.6.2 Conmon

Podman utilizza un piccolo processo di supporto chiamato **conmon** (container monitor) per gestire i container. Conmon si occupa di:

- Eseguire il container come processo separato.
- Monitorare l'output del container.
- Segnalare lo stato al client Podman.

Grazie a conmon, Podman può essere **daemonless**, perché non è necessario un daemon centrale come in Docker: ogni container ha il suo monitor.

3.6.3 Infra container

All'interno di un pod, Podman crea un **infra container** che funge da container di infrastruttura. Questo container gestisce la rete e altri namespace condivisi tra tutti i container del pod, garantendo comunicazione interna e isolamento dall'host.

3.6.4 Container init

Ogni container può avere un **container init**, che funge da processo init all'interno del container. Il container init si occupa di:

- Gestire correttamente i segnali inviati al container.
- Evitare la presenza di zombie processes.
- Avviare systemd o altri processi di init quando necessario.

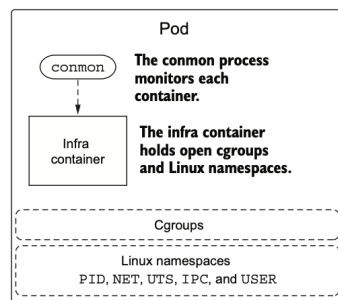


Figure 4.1 The Podman pod launches common with the infra container, which will hold cgroups and Linux namespaces.

Figura 7: Infra container: gestione della rete e dei namespace condivisi

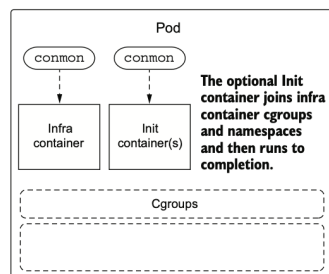


Figure 4.2 Podman next launches any init containers with common. The init containers examine the infra container and join its cgroups and namespaces.

Figura 8: Container init: gestione segnali e processi all'interno del container

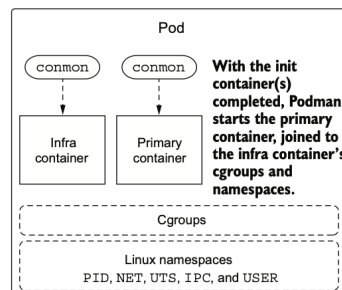


Figure 4.3 Podman waits until the init containers complete before launching the primary containers with their common into the pod.

Figura 9: Container principale: esecuzione dell'applicazione o del servizio

4 Presentazione Progetto

Il progetto realizza una piattaforma completa per l'acquisizione, la persistenza e la visualizzazione di dati in tempo reale provenienti da controllori industriali. I dati MQTT provengono da [Codesys](#), un runtime industriale utilizzato dai principali produttori di PLC. In questo caso, il runtime è eseguito in ambiente Linux su un *Raspberry Pi 5*. Codesys funge da gateway, leggendo i dati dal campo industriale (protocolli vari) e convertendoli in MQTT, un protocollo di comunicazione IoT più vicino all'ambito IT. Il progetto messo a disposizione non include il runtime Codesys originale; al suo posto viene fornito un servizio sostitutivo che simula la generazione dei dati. Questo perché, come tutti gli ambienti di sviluppo PLC, Codesys è un sistema chiuso e difficilmente integrabile con altri sistemi. Inoltre, il suo funzionamento dettagliato esula dagli obiettivi di questa relazione. L'obiettivo della piattaforma è creare un **Digital Twin** di base, in cui i dati telemetrici di un'entità, ad esempio un macchinario industriale come un pallettizzatore, vengono raccolti, monitorati e resi disponibili in tempo reale.

4.1 PLC e Industria 4.0

I **PLC** (Programmable Logic Controller) sono dispositivi industriali fondamentali per il controllo automatico di macchinari e impianti. Introdotti a partire dagli anni '70 per sostituire i sistemi a logica cablata, i PLC hanno progressivamente assunto un ruolo centrale nell'automazione industriale grazie alla loro affidabilità, robustezza e flessibilità.

Dal punto di vista del **controllo**, un PLC è progettato per:

- acquisire segnali da sensori e dispositivi di campo (ingressi digitali e analogici);
- elaborare tali segnali secondo la logica definita dal programmatore;
- comandare attuatori, motori e altri dispositivi tramite uscite digitali e analogiche.

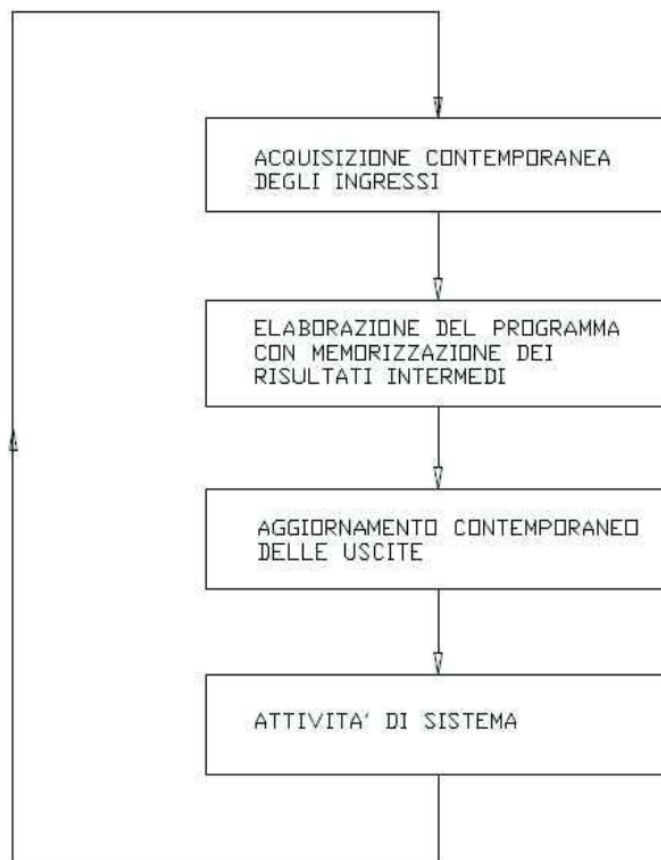


Figura 10: Ciclo PLC

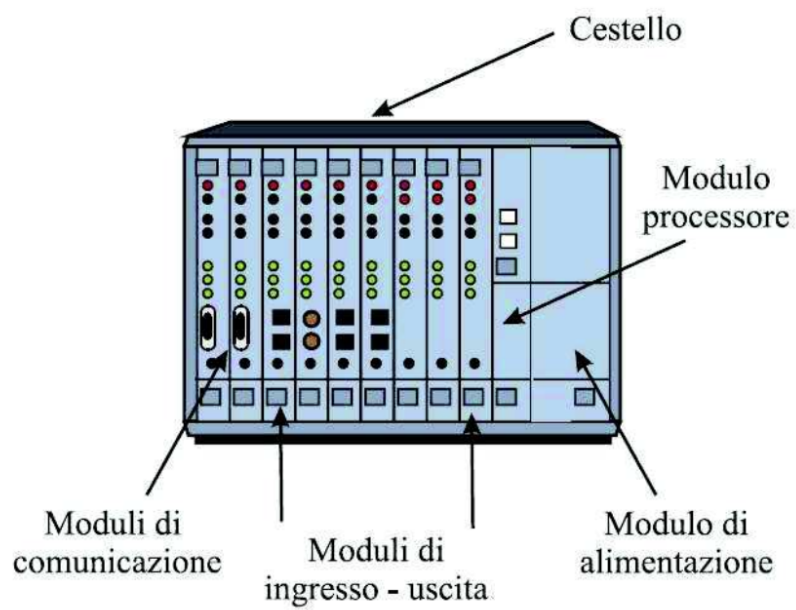


Figura 11: Schema PLC fisico



Figura 12: Tipico PLC fisico

Oggi sul mercato esistono numerosi produttori di PLC, ciascuno con le proprie caratteristiche e strumenti di sviluppo. Ogni marca fornisce un **ambiente di sviluppo integrato (IDE)** proprietario, e purtroppo molto spesso a pagamento (Codesys è gratuito), attraverso cui l'utente può programmare e configurare il dispositivo. Tra i principali produttori si possono citare:

- **Siemens** con l'ambiente *TIA Portal*;
- **Rockwell Automation** con *Studio 5000*;
- **Beckhoff** con *TwinCAT*;
- **Schneider Electric** con *EcoStruxure Control Expert*;

Nonostante la varietà di IDE, la programmazione dei PLC si basa sugli stessi **linguaggi standardizzati** definiti dalla norma IEC 61131-3, tra cui:

- **Ladder Diagram (LD)**: linguaggio grafico ispirato agli schemi elettrici a relè;
- **Structured Text (ST)**: linguaggio testuale di tipo imperativo, simile al Pascal;
- **Function Block Diagram (FBD)**: linguaggio grafico orientato a blocchi funzionali;
- **Instruction List (IL)** e **Sequential Function Chart (SFC)**, oggi meno utilizzati ma parte dello standard.

Accanto ai PLC tradizionali, esistono anche i cosiddetti **runtime PC**, come ad esempio **Codesys**. Oltre a essere adottato da numerosi costruttori di PLC con personalizzazioni proprietarie, può essere installato e utilizzato in modo indipendente come *runtime* su piattaforme generiche (es. Linux, Windows). In questo modo è possibile trasformare un comune PC o un single-board computer in un controllore programmabile, potendo affiancare al runtime tecnologie IT.

Oltre alle differenze negli strumenti di sviluppo, i PLC possono supportare diversi protocolli di comunicazione industriale, talvolta proprietari o ottimizzati per l'hardware della casa produttrice. Un ulteriore fattore di complessità è che i PLC rimangono operativi per molti anni, spesso per l'intera vita utile dell'impianto. Di conseguenza, il panorama risulta molto frammentato, con la coesistenza di generazioni diverse di dispositivi, protocolli legacy, librerie proprietarie e ambienti di sviluppo eterogenei. Questo rende l'integrazione con sistemi moderni una sfida significativa, soprattutto nell'ambito dell'Industria 4.0.

4.1.1 Protocolli di comunicazione

Nel contesto dell'automazione industriale, come detto prima, esistono numerosi protocolli di comunicazione, ognuno con funzionalità, prestazioni e livelli di astrazione differenti. Alcuni protocolli sono pensati per garantire tempi di risposta deterministici e quindi adatti al controllo in tempo reale (tipicamente nelle reti di campo), mentre altri si focalizzano su interoperabilità e scambio dati a un livello più alto. Spesso ogni casa produttrice di PLC ha spinto storicamente un proprio protocollo: ad esempio **Beckhoff** con EtherCAT, **Rockwell Automation** con EtherNet/IP, e **Siemens** con Profinet. Di seguito una panoramica dei protocolli più diffusi.

EtherCAT (Ethernet for Control Automation Technology) EtherCAT, sviluppato da Beckhoff, è un protocollo real-time basato su Ethernet. È progettato per cicli di controllo rapidi e deterministici, tipici del motion control e delle applicazioni dove il tempo di risposta è critico. Caratteristiche principali:

- Architettura master/slave: un dispositivo centrale interroga uno o più nodi.
- Sincronizzazione precisa dei nodi tramite distributed clocks e circuito ad anello chiuso.
- Comunicazione *on-the-fly*: i dati vengono letti e scritti mentre il pacchetto attraversa i nodi, riducendo drasticamente la latenza.
- Elevata scalabilità: centinaia di nodi con tempi di ciclo inferiori al millisecondo.
- Ampio supporto da parte di dispositivi di automazione e motori/drive.

EtherNet/IP (Ethernet Industrial Protocol) EtherNet/IP è promosso da Rockwell Automation e si basa sullo standard CIP (Common Industrial Protocol). È ampiamente diffuso soprattutto in Nord America. Caratteristiche principali:

- Usa Ethernet standard e TCP/UDP per il trasporto, rendendolo facile da integrare nelle reti IT.
- Supporta sia dati ciclici (real-time I/O) sia aciclici (diagnostica, configurazione).
- Maggiormente orientato al controllo di processo e comunicazione PLC-to-PLC o PLC-to-device, con meno determinismo rispetto a EtherCAT.

Profinet Profinet è il protocollo sviluppato da Siemens e lo standard di riferimento nei loro sistemi PLC. Caratteristiche principali:

- Basato su Ethernet industriale, con estensioni per garantire tempi di risposta deterministici (Profinet IRT – Isochronous Real Time).
- Forte integrazione con i PLC Siemens e i loro strumenti di configurazione (TIA Portal).
- Diffuso in Europa, soprattutto negli impianti che fanno largo uso di Siemens.

OPC UA (Open Platform Communications Unified Architecture) OPC UA rappresenta un approccio di livello superiore: è orientato alla *machine-to-machine communication* con focus su interoperabilità e sicurezza. Caratteristiche principali:

- Indipendente dall'hardware e dal vendor: consente a dispositivi di diversi produttori di comunicare senza problemi.
- Basato su TCP/IP, con supporto per crittografia, autenticazione e certificati.
- Fornisce un modello informativo strutturato: non solo valori di processo, ma oggetti con proprietà, metodi ed eventi.
- Ideale per l'integrazione con sistemi IT, MES, SCADA e per applicazioni di *Industria 4.0*.

Modbus Modbus è uno dei protocolli più longevi, introdotto da Modicon negli anni '70. Ancora oggi è molto diffuso per la sua semplicità e universalità. Caratteristiche principali:

- Architettura master/slave: un dispositivo centrale interroga uno o più nodi.
- Trasmissione tramite registri e coil che rappresentano stati e valori numerici.
- Supporto sia seriale (RS-232, RS-485) che TCP/IP (Modbus TCP).
- Utilizzato ancora oggi soprattutto in impianti legacy e in sistemi misti dove serve semplicità e compatibilità.

4.2 Architettura e Flusso dei Dati

Il runtime Codesys, eseguito su un Raspberry Pi 5, funge da gateway tra i protocolli di comunicazione industriale raccogliendo i dati dai dispositivi di campo per poi inoltrarli tramite protocollo MQTT al broker installato all'interno del Pod. I Pod sono eseguiti su un altro device, separati dal Raspberry Pi per motivi di sicurezza e prestazioni. In linea teorica, lo stesso Raspberry Pi potrebbe ospitare anche i Pod, ma mantenere il runtime Codesys isolato permette di garantire una maggiore stabilità al sistema di controllo industriale.

Il Pod è composto da quattro componenti:

- **Broker MQTT (Mosquitto):** Funge da intermediario centrale per i messaggi. I dispositivi o i simulatori pubblicano i dati su specifici *topic* (es. `palletizer1//temperature`) e il broker li distribuisce ai client sottoscritti. La configurazione in `mosquitto.conf` gestisce le connessioni e il logging.
- **Client MQTT-Database (mqtt_to_timescale):** È il cuore del sistema, un'applicazione scritta in Go (`go/mqtt_to_timescale.go`). Questo servizio si connette al broker Mosquitto, si sottoscrive a un pattern di topic (es. `palletizer1/#`) e processa ogni messaggio ricevuto. Per ogni messaggio, estrae il *payload* e lo inserisce in un database *time-series*. Il client è progettato per essere robusto, con logiche di riconnessione (`connectDB`) e worker concorrenti (`dbWorker`) per gestire carichi elevati.
- **Database Time-Series (TimescaleDB):** Scelto per l'efficienza nella gestione di grandi volumi di dati cronologici. Al primo avvio, lo script `01_init.sql` crea automaticamente la tabella `mqtt_data` e la converte in una *hypertable*, ottimizzata per query basate sul tempo. I dati vengono resi persistenti sul disco host, garantendo la sopravvivenza ai riavvii del pod.
- **Piattaforma di Visualizzazione (Grafana):** Utilizzata per creare dashboard interattive. Grafana si connette a TimescaleDB come sorgente dati e, tramite il provisioning automatico definito nella cartella `provisioning`, carica una dashboard preconfigurata che visualizza i dati raccolti.

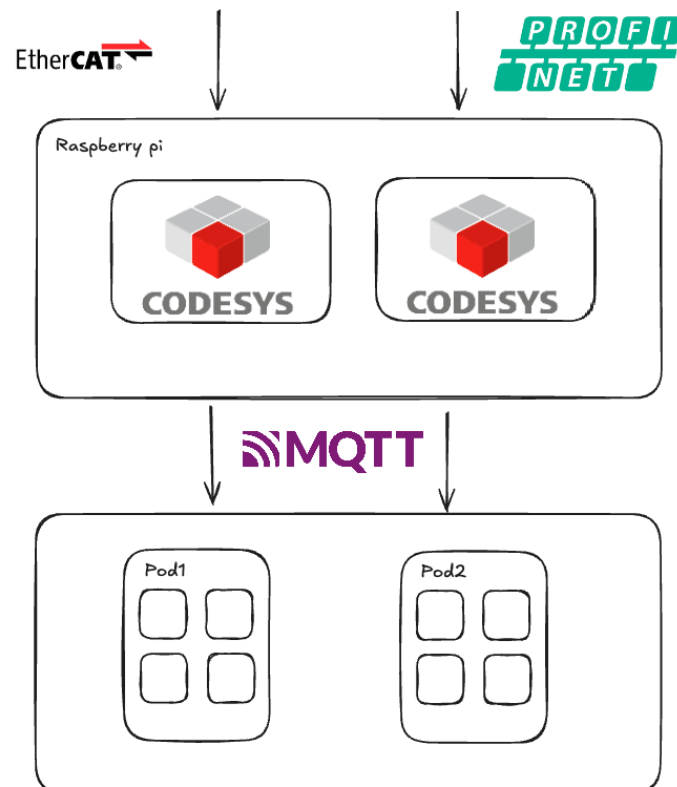


Figura 13: Flusso dati

Per ogni runtime Codesys, i segnali provenienti da diversi PLC vengono convertiti in MQTT. Ogni oggetto del macchinario da cui si raccolgono i dati è modellato come un *topic* MQTT dedicato; il *payload* pubblicato è in formato **JSON** e contiene i relativi attributi (stato, misure, identificativi, ecc.). La pubblicazione avviene con cadenza **100 ms**.

```
▼ palletizer
  ▼ Elevator
    ▼ Objects
      PalletFrontLimitSensor = { "IsActive": "1", "IsDisabled": "0", "IsForced": "0" }
      BoxClamper = { "BackwardValve": "0", "ForwardValve": "0", "IsBackward": "0", "IsForward": "0", "Alarm": "0" }
      ElevatorPlate = { "BackwardValve": "1", "ForwardValve": "0", "IsBackward": "0", "IsForward": "0", "Alarm": "0" }
      ElevatorMotor = { "Torque": "0", "Velocity": "0", "Position": "0", "Current": "0", "TempMotor": "0", "TempDrive": "0", "Command": "None", "MotionState": "Disabled", "Alarm": "0" }
      ElevatorChain = { "Torque": "0", "Velocity": "0", "Position": "0", "Current": "0", "TempMotor": "0", "TempDrive": "0", "Command": "None", "MotionState": "Disabled", "Alarm": "0" }
      PalletBackLimitSensor = { "IsActive": "1", "IsDisabled": "0", "IsForced": "0" }
    ► BoxFeeder (5 topics, 1245 messages)
    ► MainMachine (7 topics, 1740 messages)
    ► ProductInfeedConveyors (2 topics, 496 messages)
    ► PalletInfeed (2 topics, 496 messages)
    ► PalletOutfeed (2 topics, 496 messages)
```

Figura 14: Struttura Topic MQTT

Un servizio monitora il broker MQTT e, a ogni aggiornamento, storicizza il messaggio in TimescaleDB, ottimizzato per serie temporali. Lo schema logico prevede una singola tabella in cui vengono salvati *topic* e *payload* (JSON) insieme al timestamp di arrivo e ad un eventuale identificativo del pod.

Schema tabellare (TimescaleDB)

```
CREATE TABLE IF NOT EXISTS mqtt_data (
  time      TIMESTAMPTZ NOT NULL DEFAULT NOW(),
  topic     TEXT        NOT NULL,
  value     JSONB,
  pod_name  TEXT
);
```

Su Grafana è disponibile una *dashboard* preconfigurata che consente, ad esempio, di monitorare l'andamento nel tempo della posizione di un motore (segnale già predisposto nella dashboard). In questo modo si ottiene una visualizzazione immediata dell'evoluzione dei parametri di interesse a partire dai messaggi MQTT persistiti nel database.

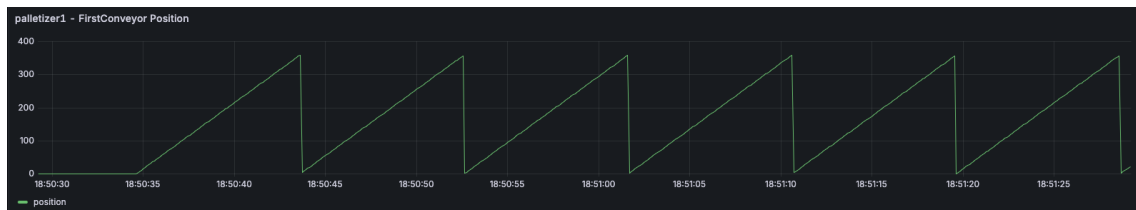


Figura 15: Posizione Motore nel tempo (Motore rotativo: da 0 a 360)

Visto che il progetto e il runtime Codesys non vengono forniti, all'interno del pod è presente un ulteriore servizio che si occupa della generazione dei dati andando a simulare la lettura tramite Codesys.

4.3 Repository Github

Il progetto è disponibile su [Github](#). Nel file Readme sono riportati i passaggi necessari per lanciare i pod. Su GitHub non è incluso il progetto Codesys, ma un eseguibile in Golang che simula la pubblicazione dei dati tramite Mqtt. Questo perché l'installazione e l'esecuzione corretta di Codesys richiedono diversi passaggi specifici della piattaforma (Gui e altre configurazioni). Se si vuole provare Codesys, allego un videotutorial che mostra come installare Codesys e avviare il runtime: [Tutorial](#). Da qui è inoltre possibile scaricare Codesys senza registrazione al sito ufficiale: [Download](#) (Finder utilizza il runtime sui suoi nuovi PLC).

4.4 Struttura del Progetto

Il progetto è organizzato in modo modulare, con una cartella dedicata a ciascun servizio principale che compone la piattaforma. Ogni cartella contiene i file di configurazione, i dati e gli script necessari al funzionamento del rispettivo componente. In particolare, sono presenti cinque directory principali:

- **mosquitto**: ospita i file di configurazione per il broker MQTT.
- **go_to_timescale**: contiene il codice e il binario del client che si occupa di ricevere i messaggi MQTT e salvarli nel database.
- **timescale**: contiene la configurazione e i dati persistenti del database time-series.
- **grafana**: include la configurazione, i dati e le dashboard per la visualizzazione dei dati raccolti.
- **go_simulation**: contiene il codice e il binario per simulare i dati in arrivo dal runtime Codesys (Solo la posizione del motore).

A supporto della gestione dell'infrastruttura, sono disponibili due script Bash che permettono di creare e rimuovere facilmente i pod, comprese le cartelle dei dati generati nel tempo dai servizi, consentendo l'esecuzione di più istanze indipendenti della piattaforma. L'intera configurazione dei container e dei volumi condivisi è centralizzata in un unico file template, che viene parametrizzato tramite script al momento dell'avvio per adattarsi a ciascuna istanza.

Inoltre, è presente un secondo file YAML privo di variabili parametriche, che consente di lanciare un pod direttamente da riga di comando senza alcuna personalizzazione.

5 Esecuzione Pods

5.1 Installazione Podman

Podman può essere installato nativamente su Linux, mentre su Windows e Mac richiede l'uso di una macchina virtuale leggera tramite **Podman Machine**.

Linux L'installazione è diretta tramite il package manager della distribuzione.

- **Ubuntu / Debian:**

```
sudo apt update
sudo apt install podman -y
podman --version
podman info
podman run hello-world
```

Windows e Mac è necessario abilitare una macchina virtuale per eseguire Podman:

1. Windows: Scaricare e installare Podman da <https://podman.io/getting-started/installation>.
MacOS: brew install podman
2. Aprire PowerShell o terminale e creare la macchina virtuale:

```
podman machine init
```

3. Avviare la macchina:

```
podman machine start
```

4. Verificare l'installazione:

```
podman info
podman run hello-world
```

Dettagli

- La macchina virtuale viene creata una sola volta e può essere riutilizzata.
- I percorsi dei volumi devono essere accessibili dalla VM (su Windows sotto `C:\Users\utente>`, su Linux/Mac sotto la home dell'utente).
- Tutti i comandi Podman su Windows e Mac vengono eseguiti all'interno della VM, ma l'uso è trasparente per l'utente.

Una volta installato podman si può verificare che effettivamente i comandi da CLI sono uguali a quelli di docker, ad esempio con `podman ps -a` si può verificare l'esistenza del container hello-world.

5.2 Differenze con Docker-Compose

L'utilizzo dei Pod in Podman tramite file di definizione in formato **YAML** (in stile Kubernetes) presenta alcune peculiarità rispetto all'utilizzo di **docker-compose** tramite file **docker-compose.yml** (PS: è possibile utilizzare docker-compose anche con podman, la CLI è sempre la stessa).

- I percorsi dei volumi devono essere assoluti: Podman non supporta i path relativi come invece fa Docker Compose. Il problema, come vedremo dopo, è aggirabile tramite il passaggio di variabili d'ambiente.
- Non è disponibile un meccanismo di *hot reload*: un cambiamento ai file di configurazione non viene automaticamente applicato ai container già in esecuzione. Per rendere effettive le modifiche è necessario eliminare e ricreare il Pod con il nuovo file di definizione, mentre in **docker-compose** è sufficiente rilanciare il comando **up** per aggiornare i servizi modificati.
- Non è supportata la direttiva **depends_on** tipica di **docker-compose**, quindi non è possibile specificare l'ordine di avvio dei container. Per gestire le dipendenze è necessario:
 - Dipendenze interne al pod: gestire l'attesa nel codice dei servizi, sfruttando il fatto che i container condividono **localhost**.
 - Dipendenze esterne al pod: utilizzare gli **init** container per attendere che i servizi esterni siano disponibili prima dell'avvio del pod.
- All'interno di un Pod tutti i container condividono lo stesso namespace di rete e possono comunicare tra loro direttamente tramite **localhost**, senza la necessità di creare reti dedicate o configurazioni aggiuntive: i servizi devono quindi essere in grado di auto-organizzarsi e "pingarsi" autonomamente, effettivamente bisogna gestirli come se girassero sulla stessa macchina senza nessun servizio di virtualizzazione. Effettivamente per accedere a dei servizi che girano sul nostro host principale ci accedo non tramite **localhost** ma tramite **host.containers.internal**.
- Entrambi gli approcci permettono di lanciare più istanze dello stesso file di definizione utilizzando variabili d'ambiente per la parametrizzazione, ma presentano differenze significative nella gestione pratica.

```
apiVersion: v1
kind: Pod
metadata:
  name: ${POD_NAME} # qui viene usata la variabile
spec:
  containers:
    - name: app
      image: myapp:latest
```

Prima di lanciare il Pod, si definisce la variabile d'ambiente:

```
export POD_NAME=my-pod1
podman play kube pod.yaml
```

Per avviare un'altra istanza, basta cambiare la variabile:

```
export POD_NAME=my-pod2
podman play kube pod.yaml
```

In `docker-compose`, invece, è possibile avviare più istanze dello stesso `docker-compose.yml` specificando un nome di progetto diverso (`-p`) o un file `.env` separato. Ad esempio:

```
# Prima istanza
INSTANCE=1 PORT=8081 docker-compose -p app1 up -d

# Seconda istanza
INSTANCE=2 PORT=8082 docker-compose -p app2 up -d
```

Tuttavia, per ottenere un vero isolamento tra le istanze, è necessario parametrizzare tutte le risorse che potrebbero essere condivise o entrare in conflitto. Il semplice cambio di nome del Pod o del progetto `docker-compose` non è sufficiente se ci sono:

- **Porte mappate sull’host:** Se più istanze tentano di utilizzare la stessa porta dell’host, si verificherà un errore di conflitto.
- **Volumi con percorsi assoluti:** Volumi che puntano allo stesso percorso dell’host saranno condivisi tra tutte le istanze.
- **Nomi di rete o volumi fissi:** Risorse con nomi hardcoded nel YAML saranno condivise.

Per esempio, un Pod completo parametrizzato dovrebbe includere:

```
apiVersion: v1
kind: Pod
metadata:
  name: ${POD_NAME}
spec:
  containers:
  - name: app
    image: myapp:latest
    ports:
    - containerPort: 80
      hostPort: ${HOST_PORT}      # Porta host parametrizzata
  volumes:
  - name: data
    hostPath:
      path: ${DATA_PATH}          # Percorso host parametrizzato
  containers:
  - name: app
    volumeMounts:
    - name: data
      mountPath: /app/data
```

E un `docker-compose.yml` parametrizzato:

```
services:
  app:
    image: myapp:latest
    ports:
    - "${PORT}:80"                # Porta parametrizzata
    volumes:
    - "${DATA_PATH}:/app/data"    # Volume parametrizzato
    environment:
    - INSTANCE=${INSTANCE}
```

La gestione delle variabili d’ambiente differisce significativamente tra i due approcci: **Docker Compose** offre supporto nativo per i file `.env`, permettendo di specificare facilmente file di configurazione separati per ogni istanza tramite l’opzione `--env-file`. **Podman**, invece, richiede approcci manuali per caricare le variabili dai file `.env`, come l’uso di `env $(cat .env.file | xargs)` o script wrapper personalizzati.

5.3 Compilazione degli eseguibili Golang

Prima di tutto bisogna compilare i due eseguibili Golang. Dopo aver installato Golang, è necessario verificare l'architettura della VM Linux. Per farlo si può lanciare un container ed eseguire:

```
podman exec <containername> uname -m
```

Ad esempio, sul mio laptop con architettura `arm64`, il risultato del comando è:

```
aarch64
```

A questo punto si può compilare un eseguibile Golang specificando il sistema operativo (sempre Linux per via della VM) e l'architettura:

```
GOOS=linux GOARCH=arm64 CGO_ENABLED=0 go build -o <nome_eseguibile> <nome_file>.go
```

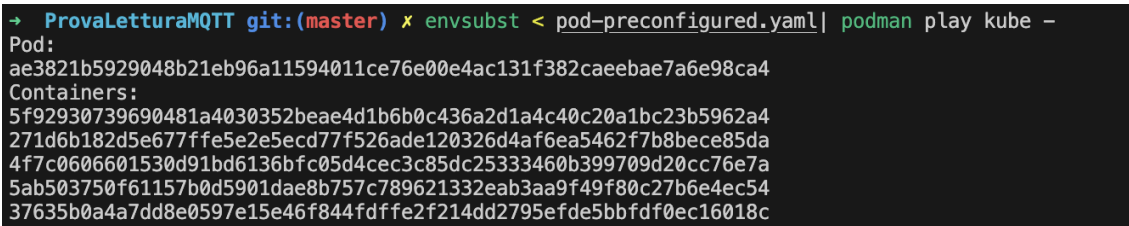
5.4 Creazione e avvio del Pod

È possibile lanciare un pod tramite:

```
podman play kube nome-pod.yaml
```

Nel progetto in esame, prima bisogna esportare la variabile `PWD`, poiché Podman non supporta i path relativi (il file `pod-preconfigured.yaml` è parametrizzato con `PWD`):

```
export PWD=$(pwd)
envsubst < pod-preconfigured.yaml | podman play kube -
```



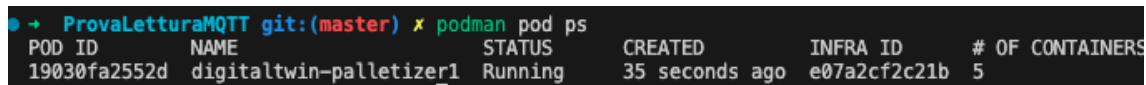
```
→ ProvaLetturaMQTT git:(master) ✗ envsubst < pod-preconfigured.yaml | podman play kube -
Pod:
ae3821b5929048b21eb96a11594011ce76e00e4ac131f382caeebae7a6e98ca4
Containers:
5f92930739690481a4030352beae4d1b6b0c436a2d1a4c40c20a1bc23b5962a4
271d6b182d5e677ffe5e2e5ecd77f526ade120326d4af6ea5462f7b8bece85da
4f7c0606601530d91bd6136bfc05d4cec3c85dc25333460b399709d20cc76e7a
5ab503750f61157b0d5901dae8b757c789621332eab3aa9f49f80c27b6e4ec54
37635b0a4a7dd8e0597e15e46f844fdffe2f214dd2795efde5bbfd0ec16018c
```

Figura 16: Risultato del lancio di un pod

5.5 Gestione dei Pod

Tutti i comandi relativi ai pod sono raggiungibili tramite `podman pod <comando>`. Ad esempio, per monitorare i pod in esecuzione:

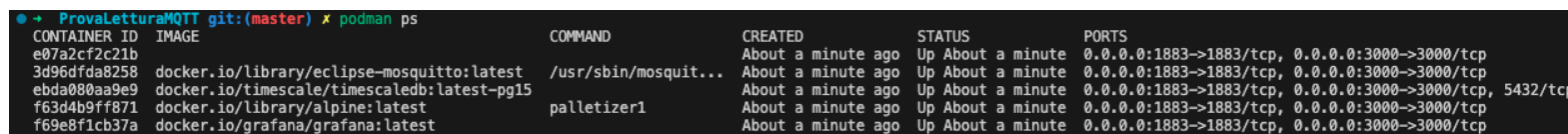
```
podman pod ps
```



POD ID	NAME	STATUS	CREATED	INFRA ID	# OF CONTAINERS
19030fa2552d	digitaltwin-palletizer1	Running	35 seconds ago	e07a2cf2c21b	5

Figura 17: Pods in esecuzione

Con il comando classico `podman ps` si possono invece vedere tutti i container eseguiti dal singolo pod. Da notare la presenza del `pod-infra`, che gestisce la comunicazione e i servizi interni al pod (il suo nome è composto da IDPod-infra).



CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
e07a2cf2c21b			About a minute ago	Up About a minute	0.0.0.0:1883->1883/tcp, 0.0.0.0:3000->3000/tcp
3d96dfda8258	docker.io/library/eclipse-mosquitto:latest	/usr/sbin/mosquitto...	About a minute ago	Up About a minute	0.0.0.0:1883->1883/tcp, 0.0.0.0:3000->3000/tcp
ebda080aa9e9	docker.io/timescale/timescaledb:latest-pg15		About a minute ago	Up About a minute	0.0.0.0:1883->1883/tcp, 0.0.0.0:3000->3000/tcp, 5432/tcp
f63d4b9ff871	docker.io/library/alpine:latest	palletizer1	About a minute ago	Up About a minute	0.0.0.0:1883->1883/tcp, 0.0.0.0:3000->3000/tcp
f69e8f1cb37a	docker.io/grafana/grafana:latest		About a minute ago	Up About a minute	0.0.0.0:1883->1883/tcp, 0.0.0.0:3000->3000/tcp

Figura 18: Container dei pod in esecuzione

5.6 Avvio, stop e rimozione dei Pod

È possibile fermare, avviare o riavviare un pod (con tutti i suoi container) con:

```
podman pod stop <nome>
podman pod start <nome>
podman pod restart <nome>
```

Per eliminare un pod:

```
podman pod rm -f <nome>
```

Lo script `clearpodfiles.sh` permette di eliminare il pod e di eliminare le sottocartelle generate (ad esempio per Timescale e Grafana), in modo da ripartire ogni volta da zero:

```
./clearpodfiles palletizer1
```

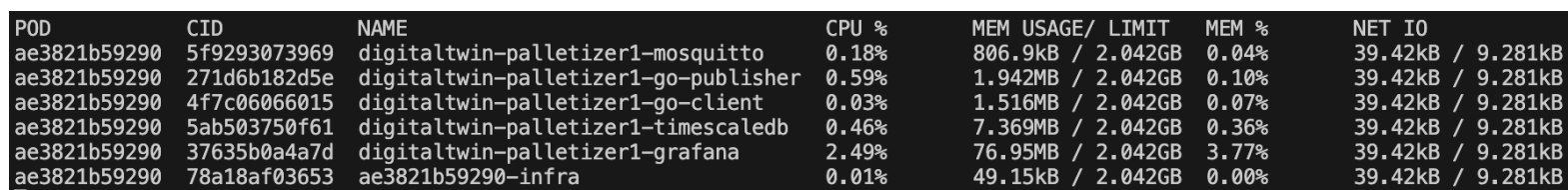
5.7 Creazione di più Pod

Per creare ed eseguire velocemente diversi pod si può utilizzare lo script `createpod.sh`. Lo script parametrizza la dashboard Grafana, e parametrizza tutti i valori del file `pod-template.yaml`. La prima porta è per il broker MQTT, la seconda per la dashboard Grafana:

```
./createpod palletizer1 1883 4000
./createpod palletizer2 1884 4001
```

Con i comandi visti in precedenza è possibile verificare che ora siano presenti due pod in esecuzione e 12 container attivi. Inoltre posso monitorare le statistiche dei container appartenenti a un pod specifico con:

```
podman pod stats <nome>
```



POD	CID	NAME	CPU %	MEM USAGE/ LIMIT	MEM %	NET IO
ae3821b59290	5f9293073969	digitaltwin-palletizer1-mosquitto	0.18%	806.9kB / 2.042GB	0.04%	39.42kB / 9.281kB
ae3821b59290	271d6b182d5e	digitaltwin-palletizer1-go-publisher	0.59%	1.942MB / 2.042GB	0.10%	39.42kB / 9.281kB
ae3821b59290	4f7c06066015	digitaltwin-palletizer1-go-client	0.03%	1.516MB / 2.042GB	0.07%	39.42kB / 9.281kB
ae3821b59290	5ab503750f61	digitaltwin-palletizer1-timescaledb	0.46%	7.369MB / 2.042GB	0.36%	39.42kB / 9.281kB
ae3821b59290	37635b0a4a7d	digitaltwin-palletizer1-grafana	2.49%	76.95MB / 2.042GB	3.77%	39.42kB / 9.281kB
ae3821b59290	78a18af03653	ae3821b59290-infra	0.01%	49.15kB / 2.042GB	0.00%	39.42kB / 9.281kB

Figura 19: Statistiche di un singolo pod

5.8 Accesso a Grafana

Infine, collegandosi alla porta esposta sul `localhost` relativa a Grafana (scelta in fase di creazione del pod), si può accedere alla dashboard preconfigurata al link:

`http://localhost:4001/dashboards`

Le credenziali di accesso di default sono:

`username: admin`

`password: admin`