

Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

# Aggiornamento e downgrade di software industriale su piattaforma Windows: analisi e soluzioni

Tesi di laurea in:  
LABORATORIO DI SISTEMI SOFTWARE

*Relatore*

**Prof. Danilo Pianini**

*Candidato*

**Francesco Foschini**

*Correlatori*

**Ing. Angelo Filaseta**

**Ing. Luigi Caiffa**

---

---

---

# Sommario

L'obiettivo di questo studio è la realizzazione di un sistema per l'aggiornamento remoto del software presente nella nuova linea di macchinari automatici sviluppati da SCM Group per la lavorazione del legno. Questo lavoro si colloca all'interno del progetto aziendale denominato *Wood-4.0*, orientato ai principi dell'industria 4.0.

Una particolare attenzione è stata dedicata al componente chiamato *PC Macchina*, presente in ogni macchina automatica. Qui risiedono gli applicativi aziendali cruciali per il corretto funzionamento di tali dispositivi e che si desidera poter aggiornare da remoto. Ad esempio, il software *HMI* consente all'operatore della macchina automatica di interagire direttamente con essa impartendo comandi.

La fase iniziale del lavoro, è stata dedicata all'analisi del contesto e allo stato dell'arte, comprendendo i concetti di *aggiornamenti Over the Air (OTA)* e le architetture per sistemi *OTA*. Sono stati esaminati casi reali in cui tali sistemi sono applicati, come nel settore automobilistico e nell'ambiente Android, per adattare le soluzioni alle esigenze delle macchine automatiche sviluppate da SCM Group.

Parallelamente, è stata condotta un'attività per esaminare le funzionalità offerte dal Package Manager Winget, selezionato per gestire le applicazioni Windows sul *PC Macchina*. Questa scelta ha contribuito alla realizzazione del sistema finale per la gestione degli aggiornamenti remoti.

L'attività è stata svolta interamente presso l'azienda SCM Group, un gruppo multinazionale italiano specializzato nella progettazione, produzione e distribuzione di macchinari e soluzioni per l'industria del legno e altri materiali. Il sistema e il codice prodotto sono stati organizzati per essere riutilizzabili nei prodotti della società e i risultati ottenuti sono stati oggetto di misurazione e presentazione.

---

---

---

*A tutti coloro che mi hanno sostenuto in questo lungo percorso.*

---

---

---

# Ringraziamenti

Vorrei esprimere il mio sincero ringraziamento al Prof. Danilo Pianini, all'Ing. Angelo Filaseta e all'Ing. Luigi Caiffa per il loro supporto durante lo sviluppo della tesi. Desidero inoltre ringraziare di cuore la mia famiglia per il costante sostegno e l'incoraggiamento che mi ha fornito nel corso di questi anni. Un ringraziamento speciale va anche a tutti i miei colleghi e amici che sono stati al mio fianco durante questo percorso.

---

---

# Indice

<b>Sommario</b>	<b>iii</b>
<b>1 Introduzione</b>	<b>1</b>
1.1 Contesto . . . . .	1
1.1.1 SCM Group e il progetto Wood-4.0 . . . . .	1
1.1.2 Contributo all'interno del progetto Wood-4.0 . . . . .	4
1.2 Motivazioni . . . . .	5
<b>2 Stato dell'arte e background</b>	<b>7</b>
2.1 Gli aggiornamenti OTA . . . . .	7
2.1.1 Architetture per sistemi di aggiornamenti OTA . . . . .	8
2.2 Applicazioni di aggiornamenti OTA . . . . .	12
2.2.1 OTA in Android . . . . .	12
2.2.2 OTA in veicoli smart . . . . .	14
<b>3 Analisi</b>	<b>17</b>
3.1 Analisi dei requisiti . . . . .	17
3.1.1 Requisiti funzionali . . . . .	17
3.1.2 Requisiti non funzionali . . . . .	18
3.2 Analisi del Problema . . . . .	18
3.2.1 Indagine comparativa di tecnologie per la gestione di appli- cazioni Windows . . . . .	18
<b>4 Design</b>	<b>27</b>
4.1 Il progetto os-auto-updates . . . . .	27
4.1.1 Come viene effettuato il deploy del software . . . . .	29
<b>5 Implementazione</b>	<b>37</b>
5.1 Ingegneria di Processo . . . . .	37
5.1.1 Repository Management . . . . .	37
5.1.2 Continuous Integration . . . . .	39

## INDICE

---

5.2	Implementazione sistema os-auto-updates . . . . .	40
5.2.1	Kotlin Multiplatform . . . . .	40
5.2.2	Programmazione asincrona . . . . .	43
5.2.3	Arrow . . . . .	45
5.2.4	Implementazione PackageURLDeploymentStrategy . . . . .	47
<b>6</b>	<b>Conclusioni e lavori futuri</b>	<b>51</b>
		<b>55</b>
	<b>Bibliografia</b>	<b>55</b>

---

# Elenco delle figure

1.1	Il centro di lavoro <i>Balestrini Power</i> per la produzione di sedie e tavoli.	3
1.2	La sezionatrice monolama automatica <i>gabbiani gt 3</i> .	4
1.3	Esempio di collegamento tra un <i>PC Macchina</i> e una una sezionatrice.	5
2.1	Architettura <i>OTA edge-to-cloud</i> .	9
2.2	Architettura <i>OTA gateway-to-cloud</i> .	10
2.3	Architettura <i>OTA edge-to-gateway-to-cloud</i> .	11
2.4	Richiami automobilistici relativi al software.	15
2.5	Rete wireless gerarchica, in cui un portale centralizzato comunica con un certo numero di veicoli nella rete.	16
4.1	Architettura del sistema con <i>os-auto-updates</i> .	28
4.2	Diagramma UML di sequenza che illustra i passaggi necessari per il deployment di un software.	30
4.3	Diagramma delle classi UML che raffigura le strategie di deployment (i campi contrassegnati con "?" sono opzionali).	32
4.4	Diagramma delle classi UML raffigurante il componente Package-Manager.	33
4.5	Diagramma delle classi UML raffiguranti i software nelle diverse fasi del deployment.	34
5.1	GitFlow.	38
5.2	Kotlin Multiplatform.	42
6.1	Integrazione con Scoop e Chocolatey.	52

ELENCO DELLE FIGURE

---

---

# Elenco dei Listings

3.1	Script powershell per la disinstallazione dell'applicativo Sour con Winget. . . . .	25
4.1	Esempio di file YAML per l'installazione di un software chiamato <i>fopl</i> . . . . .	31
5.1	Esempio di file per la <i>CI</i> su Github. . . . .	41
5.2	<i>Expected OsCommand</i> . . . . .	43
5.3	Implementazione <i>OsCommand</i> per JVM. . . . .	43
5.4	Implementazione <i>OSCommand</i> nativo. . . . .	44
5.5	Esempio di <i>suspending function</i> . . . . .	44
5.6	Esempio di utilizzo delle coroutines in Kotlin. . . . .	46
5.7	Fase di <i>fetch</i> della strategia di deployment <i>PackageURLDeploymentStrategy</i> . . . . .	49
5.8	Utilizzo di Arrow e coroutines per il download di un'eseguibile software. . . . .	50



---

# Capitolo 1

## Introduzione

Il presente lavoro è stato condotto presso l'azienda SCM Group, con sede a Rimini, specializzata nelle tecnologie per la lavorazione di una vasta gamma di materiali: legno, plastica, vetro, pietra, metallo e materiali compositi.

### 1.1 Contesto

#### 1.1.1 SCM Group e il progetto Wood-4.0

SCM Group è un gruppo multinazionale italiano specializzato nella progettazione, produzione e distribuzione di macchinari e soluzioni per l'industria del legno e altri materiali. Fondata nel 1952 a Rimini, l'azienda ha una presenza globale con filiali e distributori in tutto il mondo, con una produzione annuale di oltre 17.000 macchine e oltre 4.000 dipendenti.

SCM Group si articola in sei divisioni:

- **SCM:** tecnologie per la lavorazione del legno;
- **CMS:** tecnologie per la lavorazione di vetro, pietra, metallo, plastica, alluminio e compositi;
- **Hiteco:** meccatronica;
- **Steelmec:** carpenteria metallica;

- **ES:** settore elettrico e dell'elettronica;
- **SCMfonderie:** fusioni in ghisa.

Il progetto *Wood-4.0* mira a progettare, sviluppare e convalidare una nuova generazione di macchine per la lavorazione del legno in linea con il paradigma dell'industria 4.0 [1].

Una macchina per la lavorazione del legno può essere specifica o multi-purpose.

Le macchine specifiche [2] sono specializzate e molto efficienti in un tipo specifico di operazione di lavorazione.

Una operazione di lavorazione è un processo eseguito su un pannello grezzo di legno al fine di aggiungervi particolari dettagli o di trasformarlo. Gli esempi più comuni includono:

- **Bordatura:** consiste nell'applicare bordi decorativi o di rinforzo per creare bordi lisci e uniformi su pannelli di legno;
- **Foratura:** consiste nel praticare fori precisi e regolari su pannelli di legno, permettendo di crearne di diverse dimensioni e forme per viti, perni, bulloni e altri elementi di fissaggio;
- **Sezionatura:** consiste nel taglio simultaneo di uno o più pannelli al fine di ottenere elementi di dimensioni ridotte.

Diversamente dalle macchine specifiche, le macchine multi-purpose [3] possono svolgere diverse operazioni di lavorazione, consentendo lavorazioni complesse, ma non sono efficienti come le macchine specifiche.

L'obiettivo principale del progetto *Wood-4.0* è la creazione di due linee complete e razionali di macchine automatiche per la lavorazione del legno: i Centri di Lavoro a portale mobile <sup>1</sup> e le Sezionatrici Monolama <sup>2</sup>. Queste dovranno essere caratterizzate dall'integrazione di software completamente aggiornabile da remoto.

Attualmente, le macchine automatiche prodotte da SCM Group non dispongono di meccanismi per il downgrade del software a versioni precedenti, una funzionalità cruciale in caso di malfunzionamenti significativi. Inoltre, molti software

---

<sup>1</sup><https://www.scmgroup.com/it/scmwood/products/centri-di-lavoro.c874>

<sup>2</sup><https://www.scmgroup.com/it/scmwood/products/sezionatrici.c907>



Figura 1.1: Il centro di lavoro *Balestrini Power* per la produzione di sedie e tavoli.

fondamentali per il corretto funzionamento delle macchine automatiche dipendono fortemente da librerie sviluppate da terze parti, le quali a loro volta sono strettamente legate al sistema operativo sottostante, creando una complessa interdipendenza nota come *Dependency Hell* [4]. Nel contesto delle macchine automatiche, il sistema operativo adottato è Windows, il quale deve essere configurato per evitare gli aggiornamenti automatici dei suoi servizi dedicati. Affrontare questa situazione è essenziale poiché il vincolo imposto genera numerosi problemi, tra cui gravi preoccupazioni per la sicurezza informatica. Infatti, un software non aggiornato espone vulnerabilità note e potenzialmente sfruttabili. Attualmente, le installazioni e gli aggiornamenti del software sulle macchine automatiche avvengono manualmente tramite uno o più file di setup. L'obiettivo è eliminare completamente l'uso di tali file e privilegiare soluzioni innovative per automatizzare sia le installazioni che gli aggiornamenti del software.

I centri di lavoro a portale mobile (fig. 1.1) sono macchine multi-purpose in grado di eseguire diverse operazioni di lavorazione, come fresatura, foratura, tornitura e bordatura, all'interno di un unico sistema. Sono utilizzate per la produzione di

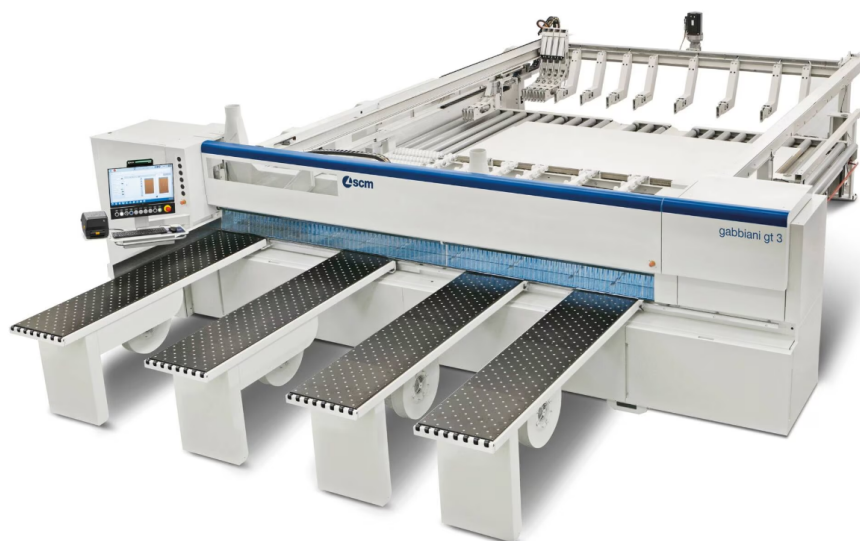


Figura 1.2: La sezionatrice monolama automatica *gabbiani gt 3*.

componenti complessi.

Le sezionatrici, invece, sono macchine specifiche progettate per eseguire l'operazione di sezionatura, ovvero il taglio preciso e regolare di uno o più pannelli contemporaneamente, al fine di ottenere pannelli di dimensioni ridotte (fig. 1.2).

Il progetto *Wood-4.0* rappresenta, quindi, un importante passo avanti nella realizzazione di una nuova generazione di macchine per la lavorazione del legno. Questo progetto costituisce il primo elemento di una roadmap di sviluppo a medio-lungo termine, che coinvolge SCM Group nell'ottimizzazione e nell'espansione della propria gamma di prodotti e servizi destinati all'industria di lavorazione del legno, in particolare nel Polo del Legno di Rimini.

### 1.1.2 Contributo all'interno del progetto Wood-4.0

Come si può notare dalla fig. 1.1 e fig. 1.2, i centri di lavoro e le sezionatrici di SCM Group sono dotati di un *PC Macchina*. Questo computer rappresenta il dispositivo collegato direttamente alla macchina fisica (centro di lavoro o sezionatrice) ed è equipaggiato con un sistema operativo Windows, su cui sono in esecuzione vari software aziendali (come illustrato in fig. 1.3). Il *PC Macchina*, in particolare, ospita l'*HMI (Human-Machine Interface)*, che rappresenta l'interfaccia attraverso

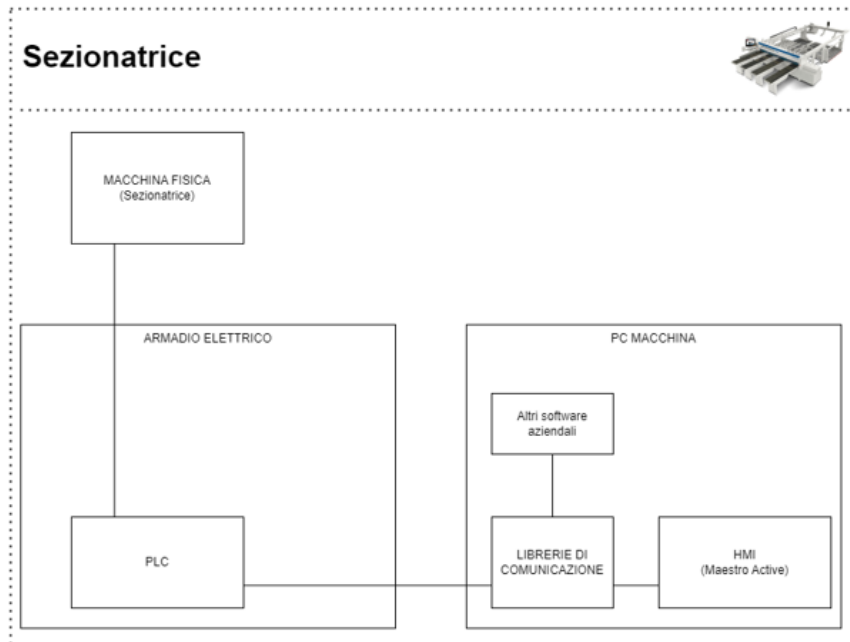


Figura 1.3: Esempio di collegamento tra un *PC Macchina* e una una sezionatrice.

la quale gli operatori possono interagire con la macchina fisica tramite un PLC (*Programmable Logic Controller*) [5]. Il PLC è un componente informatico responsabile della gestione dei processi della macchina, comunicando con i sensori analogici o digitali per il rilevamento di parametri come pressione, temperatura e posizione.

Il contributo principale all'interno del progetto *Wood-4.0*, è stato focalizzato sull'analisi e l'implementazione di pratiche per automatizzare l'installazione, l'aggiornamento e il rollback del software aziendale su sistemi operativi Windows, in particolare facendo attenzione a quelli in esecuzione sul *PC Macchina*.

## 1.2 Motivazioni

L'attività è stata condotta con l'obiettivo di perseguire le seguenti finalità aziendali, miranti a migliorare la gestione degli applicativi eseguiti sui *PC Macchina* delle nuove macchine automatiche prodotte da SCM Group:

- **Implementazione del downgrade automatico:** Sviluppare un sistema

che automatizzi la rapida e agevole reversibilità del software aziendale sulla piattaforma Windows del *PC Macchina* a una versione precedente, in caso di errori critici;

- **Implementazione dell'upgrade automatico:** Sviluppare un sistema che consenta l'upgrade automatizzato del software aziendale sulla piattaforma Windows del *PC Macchina* a una versione successiva;
- **Ridefinizione delle procedure di installazione:** Eliminare l'utilizzo dei tradizionali file di setup e adottare soluzioni innovative per semplificare le installazioni e gli aggiornamenti del software aziendale sulla piattaforma Windows del *PC Macchina*;
- **Affrontare il problema della *Dependency Hell*:** Alcuni software aziendali dipendono fortemente da librerie sviluppate da terzi, le quali a loro volta dipendono strettamente da una specifica versione del sistema operativo sottostante. In questo contesto, il sistema operativo del *PC Macchina* è Windows, il quale deve essere configurato in modo che non venga aggiornato automaticamente dai suoi servizi dedicati. Questo vincolo solleva questioni di sicurezza informatica, poiché un software non aggiornato espone vulnerabilità note.

---

# Capitolo 2

## Stato dell'arte e background

In questa sezione, è fornita un'analisi dettagliata dello stato dell'arte riguardante l'aggiornamento software da remoto per macchine industriali e le sue criticità.

Uno degli aspetti critici è la sicurezza dei dati e delle comunicazioni durante il trasferimento degli aggiornamenti [6]. È fondamentale adottare protocolli crittografici robusti per garantire la confidenzialità e l'integrità dei dati trasmessi per proteggere le macchine da potenziali attacchi informatici.

Un'altra sfida significativa riguarda la compatibilità e l'interoperabilità tra diverse versioni di software e hardware. In un contesto industriale, le macchine possono avere configurazioni hardware e software molto diverse e talvolta obsoleti. Pertanto, è fondamentale progettare un sistema di aggiornamento del software che possa gestire efficacemente questa diversità e garantire una corretta compatibilità tra le varie componenti.

### 2.1 Gli aggiornamenti OTA

Uno dei principali approcci utilizzati per l'aggiornamento del software da remoto è l'utilizzo della tecnologia *OTA* (*Over-the-Air*) [7].

Questa tecnica consente di distribuire e installare aggiornamenti software su un gran numero di macchine industriali, senza richiedere l'intervento diretto del personale tecnico sul campo.

Gli *aggiornamenti OTA* sono solitamente gestiti attraverso una piattaforma centralizzata, che consente di pianificare, monitorare e controllare l'intero processo di aggiornamento.

### 2.1.1 Architetture per sistemi di aggiornamenti OTA

Indipendentemente dal numero di dispositivi IoT [8] utilizzati da un'organizzazione, è cruciale selezionare l'approccio di *aggiornamento OTA* che meglio si adatta alle esigenze specifiche dell'azienda.

Esistono diverse architetture *OTA* progettate per garantire che i dispositivi IoT ricevano regolari aggiornamenti. La scelta dell'architettura *OTA* più adatta dipende da diversi fattori, tra cui l'hardware coinvolto, la topologia di rete, le competenze del team IT, il tipo di dispositivo IoT interessato agli aggiornamenti e i processi aziendali.

Tra le possibili architetture *OTA*, tre delle più comuni sono *edge-to-cloud*, *gateway-to-cloud* e *edge-to-gateway-to-cloud*. Ogni architettura ha le sue caratteristiche distintive e può essere adatta a contesti aziendali specifici <sup>1</sup>.

#### Edge-to-cloud

L'architettura *edge-to-cloud* (*E2C*) prevede, come mostrato in fig. 2.1, che un microcontrollore sia responsabile di recuperare gli aggiornamenti dal cloud e di applicarli al dispositivo connesso. Il microcontrollore funge da dispatcher e può essere situato direttamente sulla macchina o sui nodi ad essa connessi.

Questo tipo di architettura è comunemente utilizzata nei dispositivi smart destinati ai consumatori, poiché spesso dispongono di connettività WiFi nelle abitazioni o nei piccoli esercizi commerciali.

- **Vantaggi *edge-to-cloud*:** Gli aggiornamenti *edge-to-cloud* consentono di aggiornare i dispositivi IoT singolarmente, riducendo il rischio che un errore nell'aggiornamento influenzi l'intera flotta di dispositivi;
- **Svantaggi *edge-to-cloud*:** Non aggiornare tutti i dispositivi contemporaneamente potrebbe introdurre nuovi rischi, come la possibilità che alcuni

---

<sup>1</sup><https://www.techtarget.com/iotagenda/feature/3-OTA-architectures-for-IoT-devices>

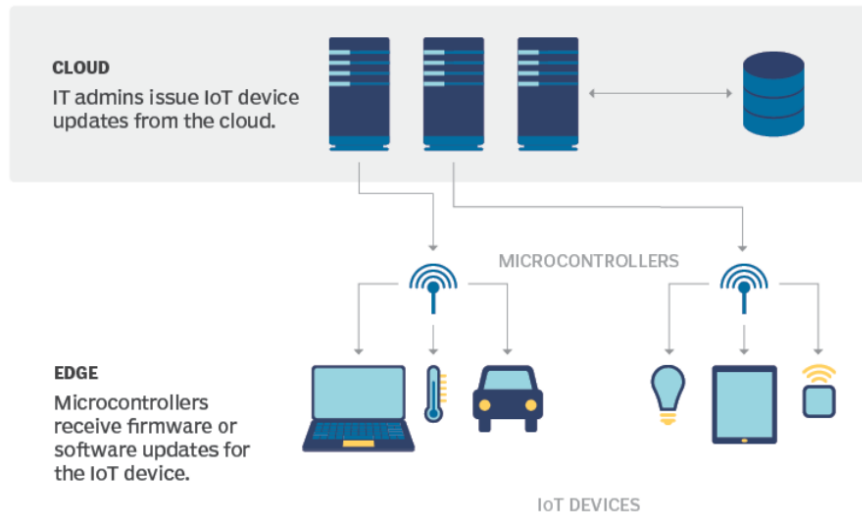


Figura 2.1: Architettura *OTA edge-to-cloud*.

dispositivi perdano aggiornamenti di sicurezza critici, rendendoli vulnerabili agli attacchi degli hacker. Potrebbero, inoltre, persistere errori hardware e software a causa della mancanza di correzioni e patch incrementali. È importante che il dispositivo non sia impegnato in attività critiche durante il processo di aggiornamento. Gli aggiornamenti sui singoli dispositivi richiedono generalmente più tempo rispetto ad altre architetture, poiché ciascun dispositivo deve essere aggiornato separatamente.

### Gateway-to-cloud

Nell'architettura *gateway-to-cloud* (*G2C*), come evidenziato in fig. 2.2, viene impiegato un *gateway* [9] per gestire un gruppo di dispositivi IoT all'interno di una rete locale.

In questo caso, i dispositivi connessi non vengono aggiornati direttamente; è invece il *gateway* stesso a essere soggetto all'aggiornamento.

- **Vantaggi *gateway-to-cloud*:** Questo sistema risulta generalmente più sicuro, poiché la potenziale superficie di attacco si limita al singolo *gateway* anziché coinvolgere tutti i dispositivi IoT. I dispositivi IoT, inoltre, non vengono toccati dagli aggiornamenti *gateway-to-cloud*, ma godono dei vantaggi

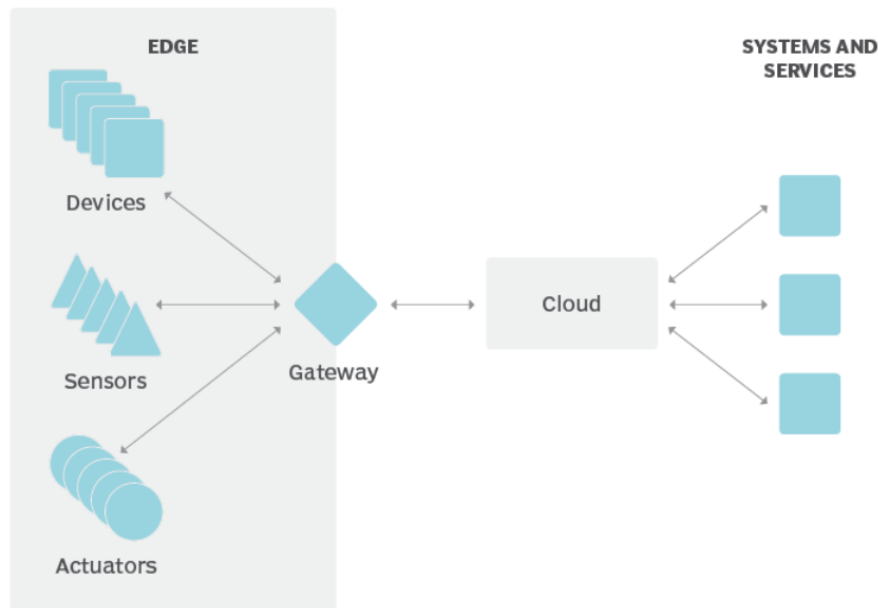


Figura 2.2: Architettura *OTA gateway-to-cloud*.

della trasmissione dei dati attraverso *gateway* sempre aggiornati. I *gateway* fungono da intermediari per connettere i dispositivi a sistemi che non soddisfano i requisiti del dispositivo stesso. Alcuni dispositivi IoT possono avere basso consumo energetico e non supportare protocolli ad alta intensità energetica come Wi-Fi o Bluetooth per connettersi direttamente al cloud. Altri dispositivi potrebbero non avere potenza di elaborazione sufficiente per gestire gli aggiornamenti o essere progettati come dispositivi monouso, non destinati a ricevere aggiornamenti;

- **Svantaggi *gateway-to-cloud*:** Il *gateway* diventa il punto critico del sistema, rappresentando un *single point of failure*. Se il processo di aggiornamento del *gateway* non viene superato con successo e non sono implementati meccanismi per il suo ripristino automatico da un aggiornamento non riuscito, il *gateway* rischia di diventare inutilizzabile, comportando persino l'interruzione dell'intero parco di dispositivi.

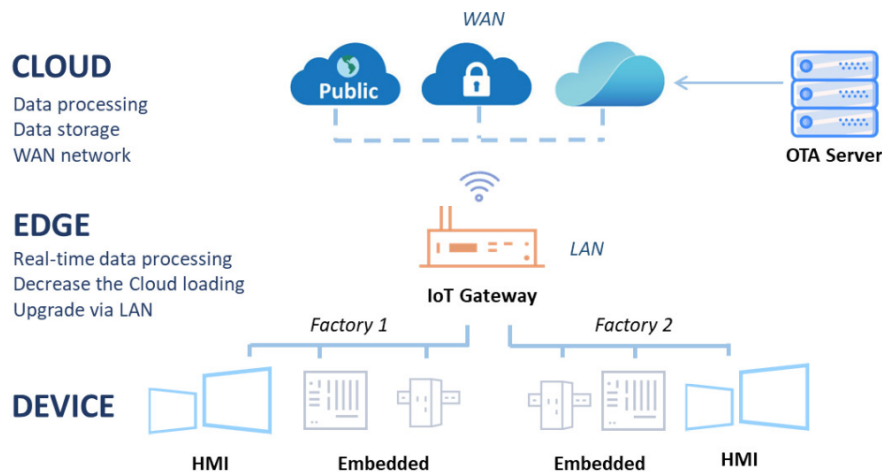


Figura 2.3: Architettura *OTA edge-to-gateway-to-cloud*.

### Edge-to-gateway-to-cloud

Nell'architettura *edge-to-gateway-to-cloud* (*E3G2C*) evidenziata in fig. 2.3, si utilizza un *gateway* connesso a Internet che controlla i dispositivi IoT e aggiorna il firmware e le applicazioni software. A differenza dello scenario *gateway-to-cloud* (*G2C*), in questa configurazione il *gateway* non viene aggiornato direttamente. Il suo ruolo è quello di agire come un dispatcher, scaricando gli aggiornamenti dal server cloud e inoltrandoli successivamente agli altri dispositivi *edge*.

- **Vantaggi *edge-to-gateway-to-cloud*:** *E3G2C* rappresenta una delle soluzioni migliori per gli *aggiornamenti OTA*. Riduce i rischi per tutti i dispositivi connessi al sistema, in quanto ogni nodo connesso al *gateway* gestisce gli aggiornamenti autonomamente. Allo stesso tempo, il *gateway* ha la capacità di inviare gli aggiornamenti direttamente su uno o più dispositivi *edge*, garantendo che eventuali problemi su un dispositivo non influenzino gli altri. Anche in presenza di uno o più nodi danneggiati, il sistema sarà in grado di continuare a funzionare;
- **Svantaggi *edge-to-gateway-to-cloud*:** Il *gateway* diventa, anche in questa architettura, un singolo punto critico per l'intero sistema. Questo può rappresentare un rischio significativo in termini di sicurezza, poiché un eventuale attacco mirato al *gateway* potrebbe compromettere l'intera rete di di-

spositivi connessi. Per mitigare questo rischio, alcune proposte sperimentali stanno esplorando l'uso della tecnologia blockchain in combinazione con smart contract per garantire l'integrità del processo di aggiornamento [10].

## 2.2 Applicazioni di aggiornamenti OTA

Gli *aggiornamenti OTA* sono diventati una pratica diffusa per la distribuzione di aggiornamenti software in molteplici ambiti. Tra i primi settori ad adottare esplicitamente questa architettura vi sono stati i telefoni cellulari.

Oltre ai telefoni cellulari, la tecnologia *OTA* si è estesa anche ai veicoli. Nel contesto dell'IoT, i settori delle macchine industriali e dei veicoli smart presentano diverse somiglianze. Entrambi i tipi di dispositivi non sono costantemente in funzione, richiedono software che interagisce con componenti a basso livello e prevedono la distribuzione di aggiornamenti tramite il cloud.

È stato condotto uno studio sugli *aggiornamenti OTA* applicati sia nel contesto Android che in quello automobilistico al fine di comprendere i principali vantaggi.

### 2.2.1 OTA in Android

Android è uno dei principali sistemi operativi mobile al mondo. Gli *aggiornamenti Over-The-Air (OTA)* sono un elemento cruciale del ciclo di vita di un dispositivo Android [11], consentendo agli utenti di ricevere nuove funzionalità, miglioramenti della sicurezza e correzioni di bug senza dover collegare il dispositivo a un computer o doverlo spedire in riparazione.

I produttori di dispositivi Android utilizzano gli *aggiornamenti OTA* per distribuire nuove versioni del sistema operativo Android stesso, oltre che per fornire patch di sicurezza mensili e aggiornamenti delle funzionalità delle app preinstallate.

Gli *aggiornamenti OTA* sono solitamente programmati per essere scaricati automaticamente in background quando il dispositivo è connesso a una rete Wi-Fi e sotto carica, per garantire che l'utente non venga interrotto durante l'utilizzo del dispositivo. gli *aggiornamenti OTA* in Android, infatti, sono progettati per essere il più possibile trasparenti e non invasivi per l'utente, consentendo loro di continuare a utilizzare il dispositivo senza interruzioni durante il processo di aggiornamento.

L'infrastruttura per gli *aggiornamenti OTA* in Android è gestita da Google tramite il Google Play Store e il servizio Google Play Services. Questo consente ai produttori di dispositivi di inviare gli aggiornamenti ai propri dispositivi Android in modo sicuro e affidabile.

Gli *aggiornamenti OTA* in Android offrono diversi vantaggi, tra cui:

- **Miglioramenti delle prestazioni:** Gli aggiornamenti possono ottimizzare il sistema operativo e le app preinstallate per migliorare le prestazioni complessive del dispositivo;
- **Sicurezza migliorata:** Le patch di sicurezza mensili distribuite tramite *aggiornamenti OTA* aiutano a proteggere il dispositivo da vulnerabilità e minacce di sicurezza;
- **Nuove funzionalità:** Gli aggiornamenti possono introdurre nuove funzionalità e miglioramenti dell'esperienza utente per mantenere il dispositivo al passo con le ultime tendenze e tecnologie.

I sistemi Android sono caratterizzati da almeno cinque partizioni principali: */system*, */cache*, */data*, */boot*, */recovery*. Ogni partizione ha un ruolo specifico e conserva tipologie di dati distinti <sup>2</sup>.

Gli *aggiornamenti OTA* su dispositivi Android sono progettati per aggiornare il sistema operativo, mantenendo invariati i dati degli utenti <sup>3</sup>. Questo avviene attraverso due modalità:

- **Aggiornamenti *Non A/B*:** Nei dispositivi più datati, l'aggiornamento avviene tramite la partizione */recovery*, la quale dispone di strumenti per decomprimere gli aggiornamenti e applicarli alle partizioni appropriate;
- **Aggiornamenti *A/B*:** I dispositivi più moderni dispongono di due copie di ogni partizione (*A* e *B*). L'aggiornamento viene applicato alla partizione non attiva, consentendo al dispositivo di scaricare e applicare l'aggiornamento durante il processo di download, evitando così la necessità di spazio di

---

<sup>2</sup>Fonte: <https://source.android.com/docs/core/architecture/partitions>

<sup>3</sup>Fonte: <https://source.android.com/docs/core/ota>

archiviazione aggiuntivo. Questo metodo di aggiornamento, rispetto al precedente, è in grado di prevenire potenziali problemi durante la procedura, poiché agisce su partizioni non attive <sup>4</sup>.

### 2.2.2 OTA in veicoli smart

I produttori automobilistici hanno introdotto sistemi di infotainment e di controllo veicolo sempre più sofisticati che richiedono aggiornamenti regolari per migliorare le funzionalità e risolvere bug del software presenti all'interno della vettura.

Un veicolo moderno contiene diverse unità di componenti elettronici (*ECU*), che svolgono compiti specifici tra cui il controllo delle funzioni dei motori, il controllo degli alzacristalli, i tergicristalli. Nell'ultimo decennio, le centraline elettroniche computerizzate hanno sostituito molti sistemi di controllo meccanico a bordo dei veicoli.

Gli *Original Equipment Manufacturer (OEM)* [12] delle diverse parti dei veicoli hanno il compito obbligato di fornire e gestire l'efficienza del software durante tutto il ciclo di vita dei veicoli. L'*OEM* svolge la funzione di garantire la correzione dei software difettosi che possono rivelarsi dannosi per quanto riguarda la sicurezza dei veicoli.

La maggior parte dei richiami alle case automobilistiche è dovuta a problemi relativi al software. Pertanto, l'*OEM* è tenuto a compiere uno sforzo supplementare per ridurre i problemi derivanti da malfunzionamenti del software. Il grafico in fig. 2.4 evidenzia la tendenza all'aumento dei richiami di veicoli associati al software. Nel 2018, 8 milioni di veicoli negli Stati Uniti sono stati affetti da una qualche forma di difetto software.

Attraverso questo esempio, diventa evidente l'importanza degli aggiornamenti software remoti *Over-The-Air (OTA)*. Molti richiami delle auto avrebbero potuto essere evitati con l'implementazione di meccanismi di *aggiornamento OTA*. Questi non solo consentono l'applicazione di patch per risolvere le vulnerabilità di sicurezza, ma sono anche cruciali per risolvere i problemi software che possono causare malfunzionamenti nei veicoli.

---

<sup>4</sup>Fonte: <https://source.android.com/docs/core/ota/ab>

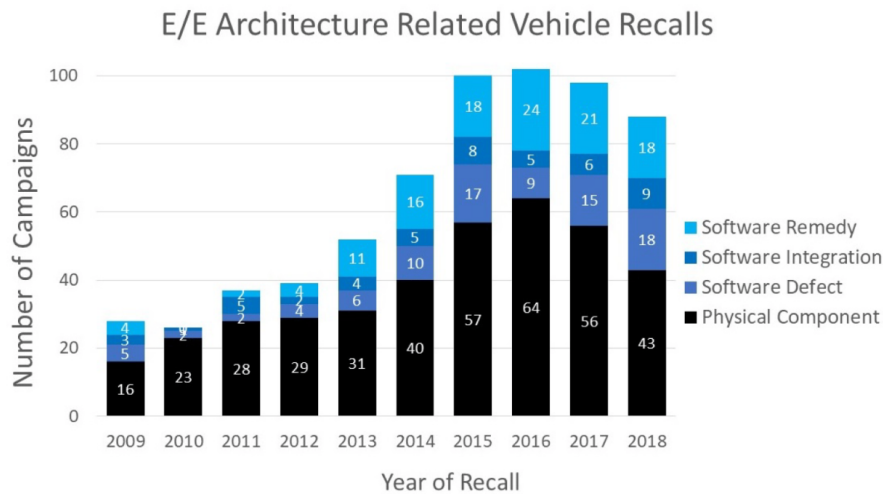


Figura 2.4: Richiami automobilistici relativi al software.

Di seguito sono elencati i vantaggi degli *aggiornamenti OTA* nel contesto dei veicoli [13]:

- **Riduzione dei richiami dei veicoli e dei costi associati;**
- **Distribuzione centralizzata del software:** Gli aggiornamenti possono essere inviati direttamente ai veicoli senza dover passare attraverso i concessionari o le officine di manutenzione (fig. 2.5);
- **Accelerazione del tempo di commercializzazione:** Il nuovo software può essere distribuito in qualsiasi momento, senza dover aspettare il ritorno del veicolo o un programma di manutenzione pianificato;
- **Maggiore convenienza:** Gli aggiornamenti possono essere effettuati ovunque e quando il cliente desidera, riducendo il tempo di inattività del veicolo;
- **Aggiornamenti obbligatori:** Il software critico per la sicurezza può essere inviato al veicolo senza richiedere l'interazione del cliente;
- **Miglioramento della sicurezza:** Gli *aggiornamenti OTA* possono ridurre il tempo di guida in condizioni non ottimali;

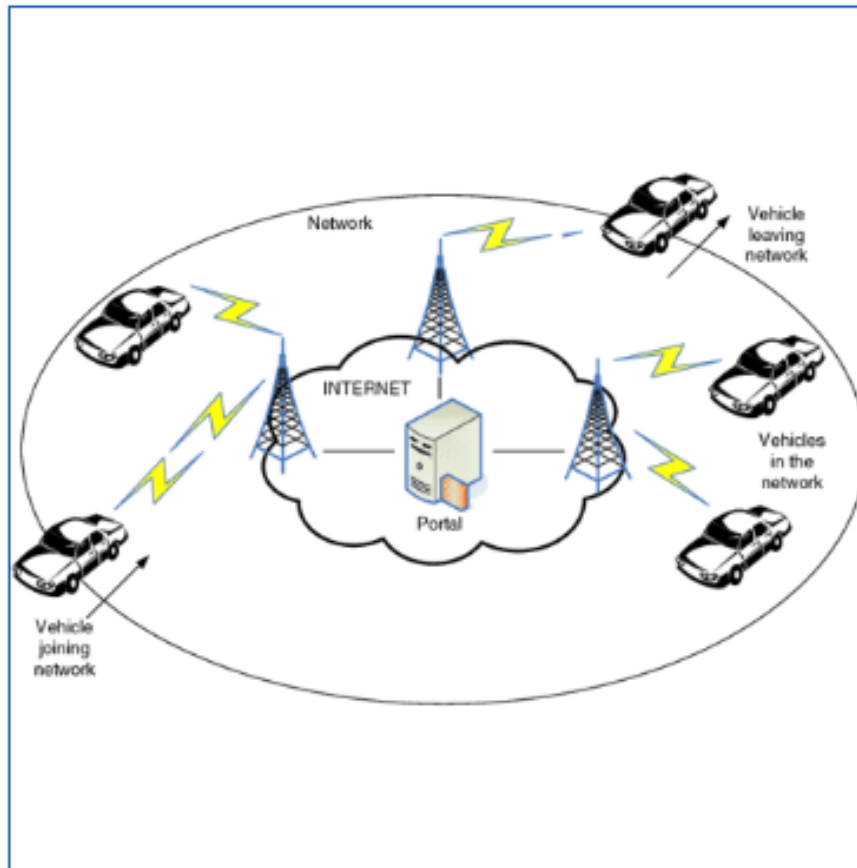


Figura 2.5: Rete wireless gerarchica, in cui un portale centralizzato comunica con un certo numero di veicoli nella rete.

- **Aumento del valore:** Mantenendo costantemente aggiornati i sistemi software, si aumenta il valore complessivo dell'auto, offrendo opportunità di guadagno aggiuntive alle case automobilistiche.

---

# Capitolo 3

## Analisi

### 3.1 Analisi dei requisiti

In questa analisi sono illustrati tutti i dettagli e requisiti che devono essere rispettati durante lo sviluppo del progetto, al fine di ottenere una soluzione che sia in linea agli obiettivi da raggiungere.

#### 3.1.1 Requisiti funzionali

I requisiti funzionali si riferiscono alle funzionalità che il sistema deve mettere a disposizione. I principali requisiti funzionali richiesti sono:

1. il sistema deve consentire la rapida e agevole reversibilità del software aziendale sulla piattaforma Windows del *PC Macchina* a una versione precedente, in caso di individuazione di bug critici;
2. il sistema deve fornire meccanismi che automatizzino l'installazione e gli aggiornamenti software sul *PC Macchina* eliminando l'utilizzo dei tradizionali file di setup;
3. Il sistema deve affrontare il problema *Dependency Hell* spiegato in sezione 1.2;
4. Il sistema deve effettuare controlli automatici per garantire l'avvenuta installazione corretta degli aggiornamenti e individuare eventuali malfunzionamenti.

menti software. In caso di errori, deve essere in grado di eseguire il rollback all'ultima versione funzionante.

### 3.1.2 Requisiti non funzionali

I requisiti non funzionali sono specifiche tecniche che definiscono le qualità che il sistema deve possedere piuttosto che le sue funzionalità specifiche. Questi requisiti sono cruciali sia per garantire un funzionamento efficiente del sistema che per fornire agli sviluppatori linee guida per una gestione ottimale del progetto. Di seguito sono elencati alcuni dei requisiti non funzionali identificati per il sistema:

1. Il sistema progettato deve essere modulare, tenendo conto della disposizione dei componenti, delle loro interazioni e della loro suddivisione;
2. La soluzione deve essere integrata nell'infrastruttura aziendale preesistente con il minimo impatto possibile;
3. Il sistema deve essere estendibile e rivisitabile;
4. Il sistema deve essere basato su tecnologie con un supporto attivo a lungo termine e che offrano nativamente una vasta gamma di aggiornamenti.

## 3.2 Analisi del Problema

### 3.2.1 Indagine comparativa di tecnologie per la gestione di applicazioni Windows

Questa sezione si focalizza sull'analisi per soddisfare gli obiettivi, analizzati nella sezione 1.2, di implementare un meccanismo per l'upgrade e il downgrade automatico dei software aziendali sul sistema operativo Windows del *PC Macchina* e ridefinire le procedure di installazione, automatizzandole e eliminando i tradizionali file di setup. Per raggiungere questi obiettivi, sono state identificate le seguenti tecnologie.

### SCCM

SCCM [14], acronimo di *System Center Configuration Manager*, è una soluzione sviluppata da Microsoft per facilitare la distribuzione, la configurazione e la gestione del software e dei dispositivi all'interno di un ambiente aziendale <sup>1</sup>.

Il funzionamento di SCCM si articola in diverse fasi:

- **Pianificazione e Configurazione:** Gli amministratori definiscono le configurazioni desiderate per i software e i dispositivi aziendali utilizzando SCCM. Questo include la creazione di pacchetti software, l'impostazione delle politiche di sicurezza e la configurazione delle regole di gestione dei dispositivi;
- **Distribuzione del Software:** SCCM consente agli amministratori di distribuire in modo centralizzato il software ai dispositivi all'interno dell'organizzazione. Utilizzando i pacchetti software precedentemente configurati, SCCM può installare e aggiornare automaticamente il software su diversi dispositivi;
- **Monitoraggio e Reporting:** SCCM fornisce strumenti per monitorare lo stato e l'utilizzo del software e dei dispositivi all'interno dell'organizzazione. Gli amministratori possono visualizzare report dettagliati sulle installazioni software, sullo stato dei dispositivi e sulle eventuali problematiche riscontrate;
- **Gestione delle Patch:** SCCM consente agli amministratori di pianificare, distribuire e monitorare le patch di sicurezza e gli aggiornamenti software su tutti i dispositivi aziendali;
- **Gestione dei Dispositivi:** SCCM offre strumenti per gestire in modo centralizzato i dispositivi all'interno dell'organizzazione. Questo include la configurazione delle impostazioni di sicurezza, il monitoraggio delle prestazioni dei dispositivi e la distribuzione delle configurazioni di rete.

Complessivamente, SCCM fornisce un'infrastruttura per la gestione del ciclo di vita del software e dei dispositivi all'interno di un'organizzazione. Essendo un

---

<sup>1</sup><https://learn.microsoft.com/en-us/mem/configmgr/core/understand/introduction>

prodotto a pagamento di Microsoft, il costo esatto può variare in base alla licenza specifica scelta e al numero di dispositivi o utenti che si desidera gestire.

SCCM è destinato principalmente a grandi organizzazioni. Potrebbe essere utilizzato, ad esempio, in un'azienda con migliaia di dipendenti distribuiti su più sedi, ciascuno con dispositivi informatici diversi e una varietà di software installato.

### SCCM Vantaggi:

- **Controllo dettagliato:** SCCM offre un controllo dettagliato sulla distribuzione del software, ideale per ambienti aziendali;
- **Supporto per software personalizzato:** Gestione di software personalizzati e proprietari senza limitazioni;
- **Ampie funzionalità:** Offre molte funzionalità oltre alla gestione del software, come la gestione delle patch e la distribuzione di immagini del sistema operativo.

### SCCM Svantaggi:

- **Complessità:** SCCM è molto complesso da configurare e richiede una certa curva di apprendimento;
- **Risorse di sistema:** Richiede risorse di sistema significative e potrebbe non essere adatto per ambienti più piccoli o utenti singoli;
- **Licenze e costi:** SCCM è un prodotto Microsoft a pagamento e può comportare costi significativi.

### Scoop

Scoop è un package manager per Windows che consente agli utenti di installare e gestire il software da riga di comando. Si basa sui seguenti principi :

- **Semplicità:** Scoop si propone di essere semplice da usare e configurare utilizzando una sintassi intuitiva da riga di comando;

- **Automazione:** Il processo di installazione dei pacchetti tramite Scoop è completamente automatizzato. Gli utenti possono eseguire singoli comandi per scaricare, installare e aggiornare i pacchetti senza dover gestire manualmente le dipendenze o utilizzare i classici file di setup per l'installazione del software;
- **Open Source:** Scoop è un progetto open-source.

Il funzionamento di Scoop è estremamente intuitivo.

Per utilizzare Scoop <sup>2</sup>, è sufficiente installarlo nel proprio sistema Windows eseguendo uno script di installazione tramite PowerShell [15], che configura automaticamente l'ambiente di Scoop sul sistema.

Una volta configurato, gli utenti possono sfruttare Scoop per cercare, installare, aggiornare e disinstallare pacchetti software disponibili nel suo repository, tutto tramite comandi semplici da riga di comando. Ad esempio, per installare un pacchetto chiamato *example*, basta eseguire il comando *scoop install example*.

Scoop, inoltre, gestisce in modo efficiente la manutenzione dei pacchetti, consentendo agli utenti di aggiornare facilmente tutti i pacchetti installati tramite il comando *scoop update*.

Grazie alla sua facilità d'uso e alla vasta gamma di funzionalità, Scoop semplifica notevolmente il processo di gestione del software su Windows, permettendo agli utenti di installare e aggiornare facilmente una vasta gamma di pacchetti direttamente da riga di comando.

### Scoop Vantaggi:

- **Leggero e facile da usare:** Scoop è leggero e ha una curva di apprendimento bassa, il che lo rende adatto per piccoli ambienti o utenti singoli;
- **Ampia raccolta di pacchetti:** Scoop offre una vasta raccolta di pacchetti già pronti per l'uso;
- **Configurazione semplice:** La configurazione di Scoop è semplice, e offre personalizzazioni in base alle esigenze.

---

<sup>2</sup><https://scoop.sh/>

### Scoop Svantaggi:

- **Limitato al software open source:** Scoop è principalmente destinato al software open source. Potrebbe essere necessario considerare altre soluzioni se si deve distribuire software proprietario..

### Chocolatey

Chocolatey è anch'esso un gestore di pacchetti per Windows che consente agli utenti di installare, aggiornare e gestire facilmente software da riga di comando. Chocolatey ha guadagnato popolarità grazie alla sua capacità di offrire agli utenti un'esperienza simile a quella disponibile su sistemi operativi basati su Linux, come apt o yum.

Chocolatey si basa sugli stessi principi di Scoop elencati e spiegati in precedenza: semplicità, automazione e open-source (sezione 3.2.1).

Per utilizzare Chocolatey è necessario prima installarlo nel proprio sistema Windows. Ciò può essere fatto eseguendo uno script di installazione, il quale configura l'ambiente di Chocolatey sul sistema.<sup>3</sup>

Una volta configurato, gli utenti possono utilizzare Chocolatey per cercare, installare, aggiornare e disinstallare pacchetti software utilizzando semplici comandi da riga di comando. Ad esempio, per installare un pacchetto chiamato *example*, l'utente potrebbe eseguire il comando *choco install example*.

Chocolatey gestisce anche la manutenzione dei pacchetti, consentendo agli utenti di aggiornare facilmente tutti i pacchetti installati tramite il comando *choco upgrade all*.

Uno dei vantaggi principali di Chocolatey è la sua flessibilità e scalabilità: può essere utilizzato sia a livello individuale che in ambienti aziendali su larga scala, consentendo agli amministratori di sistema di distribuire e gestire il software su centinaia o migliaia di computer.

### Chocolatey Vantaggi:

- **Ampia raccolta di pacchetti:** Chocolatey ha nel suo repository una raccolta di pacchetti ancora più ampia di Scoop;

---

<sup>3</sup><https://chocolatey.org/install>

- **Strumenti di creazione di pacchetti:** Consente la creazione e la gestione di pacchetti software personalizzati, sia per software open-source che per software privato;
- **Flessibilità e scalabilità:** Può essere utilizzato sia individualmente che in ambienti aziendali su larga scala.

### Chocolatey Svantaggi:

- **Sicurezza:** Poiché Chocolatey consente di scaricare pacchetti da repository di terze parti, c'è un rischio potenziale di sicurezza associato. È importante verificare attentamente la fonte dei pacchetti per garantire che siano sicuri e affidabili;
- **Risorse di sistema:** Richiede risorse di sistema più elevate rispetto a Scoop.

### Winget

Winget è un altro package manager progettato per semplificare la gestione delle applicazioni su computer Windows 10 e Windows 11. Winget offre agli utenti un metodo semplice per individuare, installare, aggiornare e rimuovere le applicazioni sul proprio sistema tramite comandi da eseguire direttamente dal terminale.

La caratteristica distintiva di Winget è la sua integrazione nativa con il servizio Gestione pacchetti Windows, il che consente agli utenti di accedere a un vasto catalogo di applicazioni disponibili nel Microsoft Store direttamente da linea di comando, evitando loro di dover cercare manualmente su Internet gli installer degli applicativi.

Essendo una soluzione nativa di Windows, per utilizzare le funzionalità di Winget da riga di comando non è necessaria la fase di installazione come nel caso dei due gestori di pacchetti analizzati in precedenza (Scoop e Chocolatey). Ad esempio, per installare un pacchetto chiamato *example* disponibile sul Microsoft Store, l'utente può eseguire il comando *winget install example*.

Grazie alla sua facilità d'uso e all'integrazione nativa con Windows, Winget offre agli utenti una strategia conveniente per mantenere aggiornato e organizzato il proprio ambiente Windows.

### Winget Vantaggi:

- **Integrato con Windows:** Essendo sviluppato da Microsoft, Winget è integrato direttamente nel sistema operativo, garantendo una migliore stabilità e compatibilità rispetto ai package manager analizzati in precedenza;
- **Vasta raccolta di pacchetti:** La sua integrazione con Microsoft Store garantisce un accesso a una vasta gamma di applicazioni disponibili per Windows.

### Winget Svantaggi:

- **Limitazioni nella personalizzazione:** Winget offre meno opzioni di personalizzazione per i pacchetti relativi alle applicazioni private rispetto a Chocolatey (come spiegato in seguito);
- **Dipendenza da Microsoft Store:** Poiché Winget utilizza il Microsoft Store per distribuire alcuni pacchetti, è soggetto alle restrizioni e alle politiche di distribuzione di Microsoft.

### Utilizzo di Winget per la gestione degli applicativi aziendali su Windows

In parallelo all'analisi descritta precedentemente sulle possibili tecnologie da adottare per la gestione degli applicativi su Windows, sono state eseguite attività sul package manager Winget al fine di comprendere appieno le sue funzionalità principali accessibili tramite linea di comando.

A tale scopo, è stata presa in considerazione un'applicazione privata sviluppata internamente da SCM Group chiamata *Sour*. Sono stati implementati script PowerShell che utilizzano Winget al fine di installare, aggiornare e disinstallare *Sour* su un pc con Windows 10. Nel listing 3.1 è riportato l'esempio relativo alla disinstallazione.

L'eseguibile dell'applicativo è stato recuperato e scaricato da linea di comando tramite script PowerShell direttamente dal repository privato di artefatti adottato da SCM Group, Sonatype Nexus Repository <sup>4</sup>. Successivamente al download, sono stati eseguiti comandi PowerShell per effettuare l'installazione di *Sour*.

---

<sup>4</sup><https://www.sonatype.com/products/sonatype-nexus-repository>

Listing 3.1: Script powershell per la disinstallazione dell'applicativo Sour con Winget.

```
1 #ottenere elenco delle applicazioni installate nel sistema
2 $appList = winget list
3
4 # Espressione regolare per trovare le applicazioni che iniziano con "SOUR-"
5 $pattern = "^SOUR-"
6
7 #Scorrere elenco delle applicazioni
8 foreach ($app in $appList) {
9     if ($app -match $pattern) {
10         # Estrazione del nome completo con versione applicazione
11         $appName = $app -split '\s+' | Select-Object -First 1
12         # Eseguire il comando di disinstallazione
13         winget uninstall $appName
14     }
15 }
```

Durante l'attività volta all'installazione di *Sour*, è emerso che Microsoft fornisce una guida dettagliata sui passaggi necessari per la creazione di pacchetti per le proprie applicazioni e per ospitarle direttamente sui propri repository <sup>5</sup>.

Dopo essere stati caricati su tali repository, i pacchetti delle applicazioni diventano visibili sul Microsoft Store. Di conseguenza, sarà possibile utilizzare i seguenti comandi Winget da linea di comando per interfacciarsi ai nuovi pacchetti, semplificando notevolmente il processo di distribuzione:

- **winget search**: per la ricerca della disponibilità del pacchetto nel Microsoft Store;
- **winget install**: per installare il pacchetto.

È emerso, tuttavia, che Microsoft non offre le stesse funzionalità per la creazione di pacchetti per applicazioni che devono rimanere private, come *Sour*. Attualmente, non esiste una strategia standard per ospitare pacchetti su repository privati, e Microsoft, per un ambiente di produzione, offre solo alcune soluzioni a pagamento <sup>6</sup>.

La decisione di adottare Winget come gestore di pacchetti per il progetto è stata presa dopo un'attenta valutazione delle opzioni disponibili e analizzate in

---

<sup>5</sup><https://learn.microsoft.com/en-us/windows/package-manager/package/repository>

<sup>6</sup><https://winget.pro/>

precedenza. Considerando la sua semplicità d'uso, l'integrazione nativa con Windows e l'ampia selezione di pacchetti disponibili su Microsoft Store, Winget si è rivelato la scelta più idonea per soddisfare le esigenze del progetto e garantire una distribuzione efficiente del software.

Questa integrazione con il Microsoft Store offre un metodo standardizzato e conveniente per distribuire le applicazioni alle macchine di SCM Group.

Nonostante al momento offra poche opzioni di configurazione per gli applicativi privati si è ritenuto che, essendo un package manager nuovo, possa ricevere un maggiore supporto nel tempo, soprattutto considerando la sua natura nativa e proprietaria di Windows. Ciò potrebbe portare a un miglioramento delle sue funzionalità e a una più ampia adozione in futuro.

Gli applicativi privati sviluppati internamente dall'azienda non saranno resi disponibili nei repository Microsoft. Attualmente, è stata presa la decisione di mantenere gli artefatti relativi agli applicativi privati nel repository manager Nexus aziendale privato e ad uso interno. Per questi applicativi, Winget sarà utilizzato esclusivamente per la disinstallazione tramite il comando *winget uninstall* e per verificare la presenza dell'applicativo tra quelli attualmente installati nel sistema tramite il comando *winget list*.

Per quanto riguarda le applicazioni open-source disponibili sul Microsoft Store e utilizzate nel contesto aziendale, invece, Winget sarà utilizzato a pieno regime, compresi i comandi di installazione tramite *winget install* e di ricerca della disponibilità tramite *winget search*.

---

# Capitolo 4

## Design

In questa sezione è analizzato il design del sistema sviluppato, chiamato *os-auto-updates*, su cui è stata effettuata una collaborazione, evidenziando anche l'integrazione delle scelte precedenti all'interno del contesto finale.

### 4.1 Il progetto *os-auto-updates*

Si è contribuito all'implementazione del sistema noto come *os-auto-updates*, concepito per soddisfare gli obiettivi inizialmente delineati nella sezione 1.2.

Il sistema è stato sviluppato per raggiungere gli stessi obiettivi presentati in questo lavoro relativi a Windows anche per il sistema operativo Linux [16]. MacOS [17] è stato escluso dallo scope del progetto.

Il sistema progettato con *os-auto-updates* si basa, come mostrato in fig. 4.1, su un'architettura simile alla *edge-to-gateway-to-cloud (E3G2C)* precedentemente analizzata nell'ambito degli *aggiornamenti software OTA* nella sezione 2.1.1. *os-auto-updates* opera a livello *edge* anziché a livello *gateway*, poiché al livello *gateway* sarà presente un servizio ancora in fase di sviluppo.

Ogni *PC Macchina* associato a un centro di lavoro o a una sezionatrice di SCM Group sarà dotato di *os-auto-updates* in esecuzione. Il principale compito di *os-auto-updates* è quello di ricevere gli aggiornamenti disponibili dall'applicativo *gateway* sotto forma di file YAML e applicarli direttamente al *PC Macchina* su cui opera. L'*applicativo gateway*, invece, funge da dispatcher scaricando gli aggiorna-

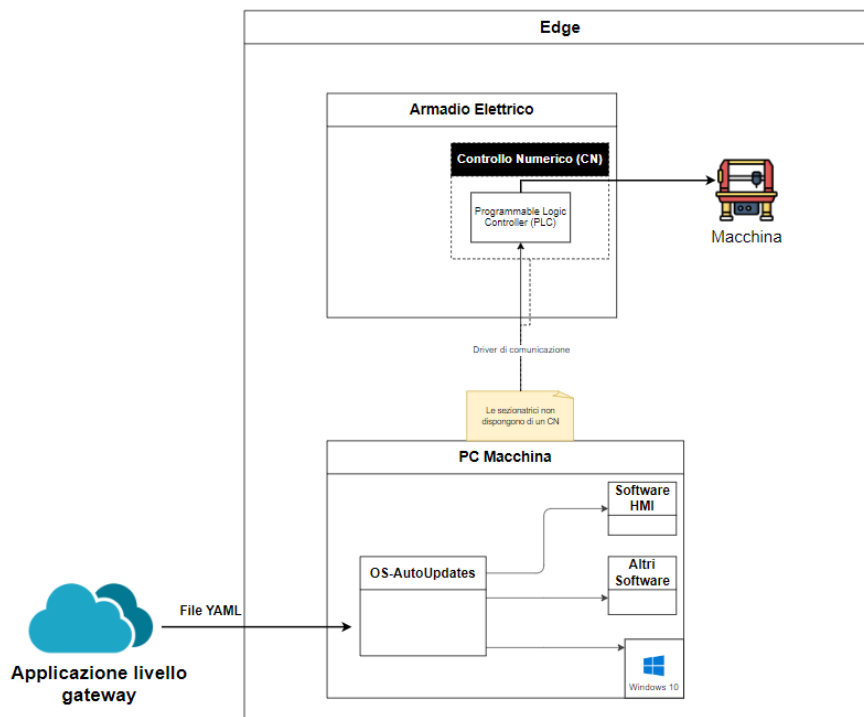


Figura 4.1: Architettura del sistema con *os-auto-updates*.

menti disponibili e inoltrandoli successivamente a tutti i *PC Macchina* su cui è in esecuzione *os-auto-updates*.

Il sistema progettato, grazie alla combinazione dell'applicativo a livello *gateway* e di *os-auto-updates*, gestisce la distribuzione degli *aggiornamenti software OTA* per Windows e Linux, sia per le applicazioni che per il sistema operativo. Il sistema garantirà quindi ai nuovi centri di lavoro e alle nuove sezionatrici prodotte da SCM Group gli stessi vantaggi discussi nella sezione 2.2.1 per gli *aggiornamenti software Over-The-Air (OTA)* su Android e nella sezione 2.2.2 per gli *aggiornamenti OTA* nei contesti dei veicoli. Ad esempio, in caso di un guasto software nel *PC Macchina* di un centro di lavoro, il cliente proprietario non sarà costretto a portare la macchina in azienda per riparare il guasto. Sarà il sistema stesso a gestire e risolvere l'errore attraverso un *aggiornamento software OTA*.

#### 4.1.1 Come viene effettuato il deploy del software

Le scelte illustrate in precedenza sono state integrate per definire l'architettura del sistema *os-auto-updates* e stabilire i passaggi necessari al deploy del software, mostrati nel diagramma UML [18] in fig. 4.2, sul *PC Macchina* delle nuove gamme di centri di lavoro e sezionatrici aziendali.

Il processo di deploy di un software tramite *os-auto-updates* segue le istruzioni contenute in un file in formato YAML ricevuto dall'applicativo *gateway*, il quale ne definisce le caratteristiche. Di seguito, nel listing 4.1, è presentato un esempio della struttura di questo file.

Per ciascun software elencato nel file YAML, incluse le dipendenze relative al software principale, vengono specificate le seguenti caratteristiche: nome del software, strategia di deployment, versione e dipendenze.

Ogni software adotta una strategia di deployment che definisce come deve essere installato, aggiornato e disinstallato. Affinché un software venga installato correttamente, deve superare le seguenti quattro fasi definite in ogni strategia di deployment:

- **fetch**: Recupero delle risorse necessarie per l'installazione del software;
- **validation**: Esecuzione di test di accettazione sul software;

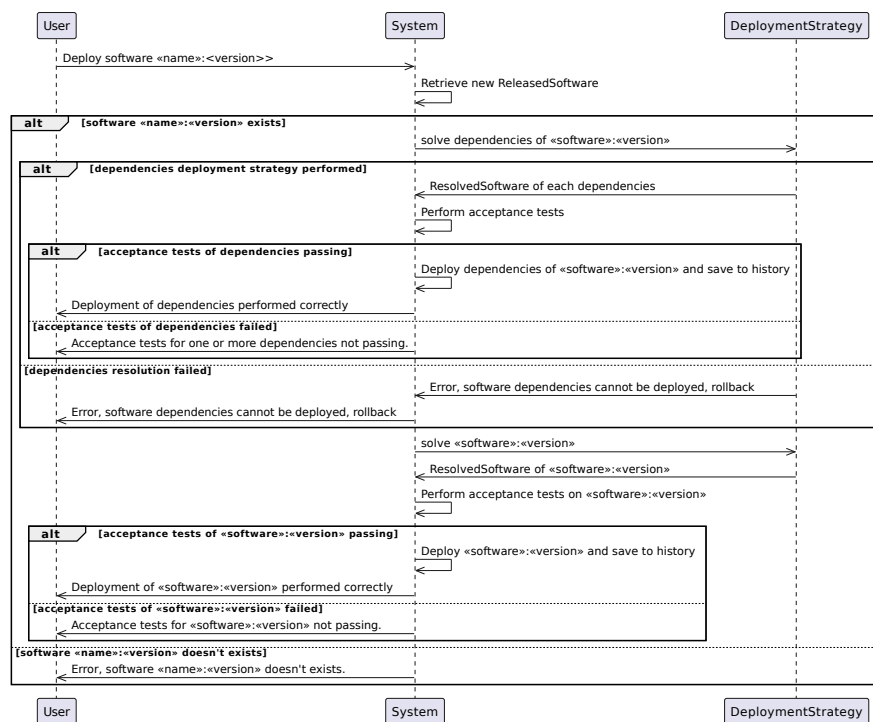


Figura 4.2: Diagramma UML di sequenza che illustra i passaggi necessari per il deployment di un software.

Listing 4.1: Esempio di file YAML per l'installazione di un software chiamato *fopl*.

```
1 name: fopl
2 deployWith:
3   type: PackageManager
4   packageManager:
5     type: Pip
6 version:
7   type: Single
8   version: "0.0.2"
9 dependencies:
10  - name: Python.Python.3.11
11    deployWith:
12      type: PackageManager
13      packageManager:
14        type: WinGet
15    version:
16      type: Single
17      version: "3.11.7"
18    dependencies: []
19  - name: cURL.cURL
20    deployWith:
21      type: PackageManager
22      packageManager:
23        type: WinGet
24    version:
25      type: Single
26      version: latest
27    dependencies: []
```

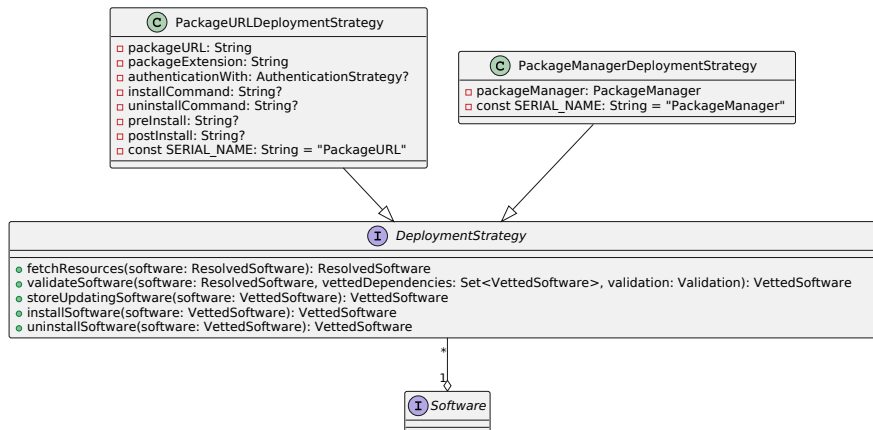


Figura 4.3: Diagramma delle classi UML che raffigura le strategie di deployment (i campi contrassegnati con "?" sono opzionali).

- **storing**: Salvataggio del software testato nello storico (in uno storage);
- **installing**: Installazione del software sulla macchina.

Le strategie di deployment, come evidenziato in fig. 4.3, prevedono anche uno step opzionale di disinstallazione nel caso di fallimenti.

Sono state definite diverse strategie di deployment, tra cui la *PackageManagerDeploymentStrategy*, la quale si basa sull'impiego di un package manager di Windows o anche su uno per Linux, che è stato utilizzato nel sistema.

Nell'esempio precedente, mostrato nel listing 4.1, sia il software *fopl* che le sue dipendenze seguono la strategia di deployment *PackageManagerDeploymentStrategy*. Mentre *fopl* utilizza Pip come package manager, tutte le dipendenze utilizzano Winget.

In questa strategia, le quattro fasi di *fetch*, *validation*, *storing* e *installing* sono implementate sfruttando i principali comandi offerti dal package manager adottato.(fig. 4.4). Nel caso di Winget, sono stati utilizzati i seguenti comandi:

- **winget search**: per cercare la disponibilità di un software nei repository di Microsoft;
- **winget list**: per elencare i software attualmente installati nel sistema;
- **winget install**: per l'installazione di un software;

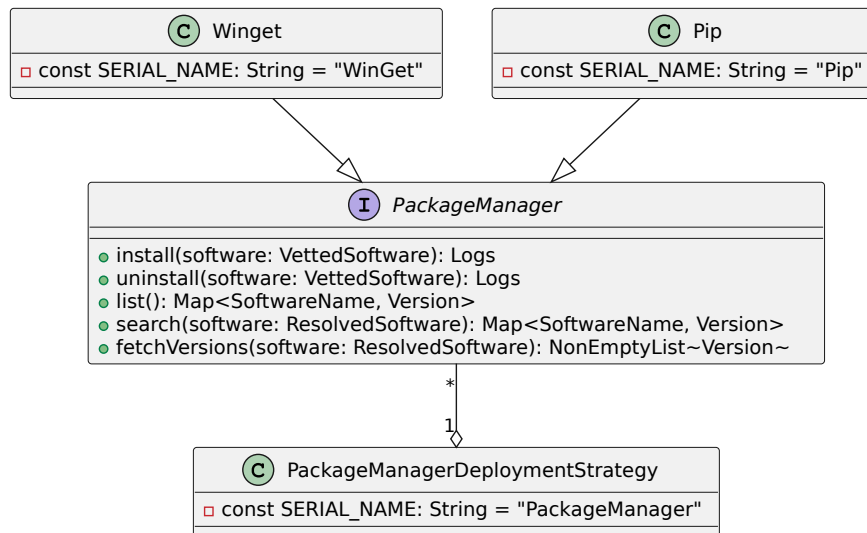


Figura 4.4: Diagramma delle classi UML raffigurante il componente PackageManager.

- **winget uninstall**: per la disinstallazione di un software.

Il processo di deploy di un software inizia con l'analisi e la deserializzazione del file YAML nel componente di sistema denominato *ResolvedSoftware*.

Un'istanza di *ResolvedSoftware*, come illustrato nel diagramma UML delle classi in fig. 4.5, mappa le principali caratteristiche di un software estratte dal file YAML, tra cui il nome, i test di validazione da effettuare su di esso, la strategia di deployment adottata, la versione e l'insieme delle dipendenze software necessarie per il suo corretto funzionamento nel sistema.

Ogni dipendenza software specificata nel file YAML viene considerata come un software autonomo e anch'essa viene mappata dal sistema in un'istanza di *ResolvedSoftware*.

Inizialmente, *os-auto-updates* tenta di installare tutte le dipendenze software specificate nel file YAML. Per ognuna di esse, vengono eseguite le quattro fasi della strategia di deployment adottata, trattando ciascuna dipendenza come se fosse un software indipendente da installare. È importante notare che anche le dipendenze seguono una strategia di deployment che deve essere attentamente rispettata, come mostrato nell'esempio di file YAML fornito nel listing 4.1

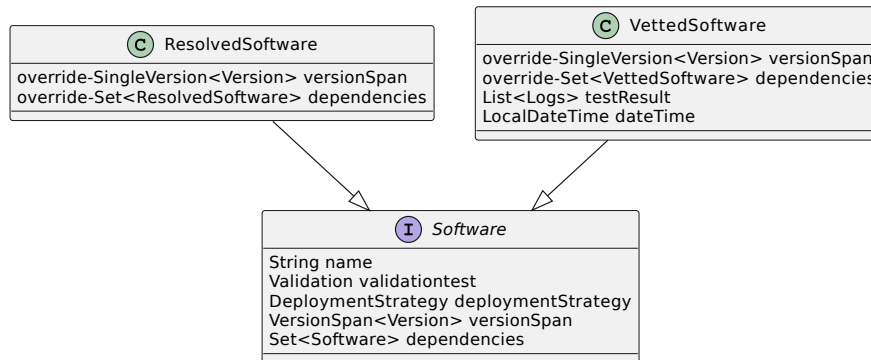


Figura 4.5: Diagramma delle classi UML raffiguranti i software nelle diverse fasi del deployment.

Indipendentemente dal fatto che un software sia già presente o meno nel sistema, il processo cercherà di installare la versione specificata nel file YAML. Se il software è già presente nel sistema con una versione diversa, sarà eseguito un upgrade o un downgrade alla versione specificata nel file.

È importante evidenziare che la strategia di deployment definisce nella fase *validation* dei test di accettazione [19] per verificare il software all'interno del sistema. Nel caso delle applicazioni Windows, verrà valutata l'usabilità nella specifica versione di Windows adottata dal *PC Macchina*. Il software che supera con successo questi test passa quindi dallo stato di *ResolvedSoftware* a *VettedSoftware*. Un *VettedSoftware* rappresenta il componente di sistema che descrive un software su cui sono stati correttamente eseguiti e superati i test di validazione, come evidenziato nel diagramma UML delle classi in fig. 4.5.

In caso di errori durante il processo di deploy di un software, ad esempio durante le quattro fasi, o nel caso in cui un'istanza di *ResolvedSoftware* non superi i suoi test di validazione, il processo verrà interrotto e si verificherà una delle seguenti situazioni:

- Se il software o alcune dipendenze erano precedentemente presenti nel sistema con una versione diversa, sarà eseguito un rollback per riportarle alla versione iniziale recuperandole dallo storico di sistema.
- Se il software e le sue dipendenze non erano installati prima dell'inizio del processo di deployment, saranno disinstallate tutte le dipendenze che erano

state installate fino al momento dell'errore per ripristinare il sistema allo stato iniziale.

Se i test di validazione di un software hanno avuto esito positivo, si potrà procedere con l'installazione e il salvataggio del software nello storico di sistema durante la fase di storing.



---

# Capitolo 5

## Implementazione

Questo capitolo inizia presentando le tecniche DevOps utilizzate per gestire lo sviluppo del progetto. Successivamente, descrive i dettagli di implementazione e le principali tecnologie utilizzate per realizzare *os-auto-updates*.

### 5.1 Ingegneria di Processo

Durante lo sviluppo del progetto *os-auto-updates*, il team ha adottato diverse tecniche di DevOps [20] per garantire un flusso di lavoro efficiente e una buona qualità del codice.

Le tecniche di DevOps sono essenziali nello sviluppo di un progetto per diverse ragioni. Da un lato, consentono di migliorare la qualità del codice mantenendo il lavoro organizzato, promuovendo il testing e garantendo l'integrazione continua dei vari componenti durante lo sviluppo, mentre tengono traccia delle versioni rilasciate. Dall'altro lato, semplificano il mantenimento di un'elevata produttività riducendo i tempi di inattività e automatizzando compiti ripetitivi, che possono essere eseguiti più efficientemente da un computer che da un essere umano, liberando così più tempo per l'implementazione delle funzionalità critiche.

#### 5.1.1 Repository Management

Il codice prodotto è stato gestito utilizzando il rinomato sistema di controllo delle versioni decentralizzato (*DVCS*) git [21], facendo leva specificamente sul servizio

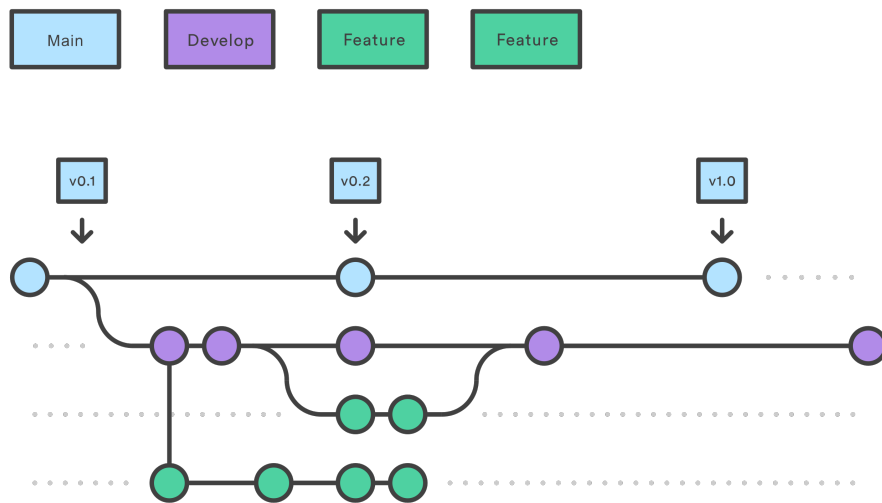


Figura 5.1: GitFlow.

di hosting GitHub <sup>1</sup>.

Per garantire un processo di sviluppo standardizzato e coerente, evitando errori e problemi di compatibilità, è stata scelta la metodologia GitFlow. GitFlow è un modello di branching che prevede l'uso di due branch principali: *master* e *develop* (fig. 5.1). *Master* è utilizzato per i rilasci, mentre *develop* è utilizzato per lo sviluppo in corso. Inoltre, per ogni nuova funzionalità richiesta viene creato un branch denominato *feature/nome-della-funzionalità*, che sarà poi unito al branch *develop* una volta completato lo sviluppo della funzionalità.

Prima di unire le modifiche al branch principale (*main/master*), le nuove funzionalità proposte da un programmatore sono state esaminate e verificate da tutti gli altri membri del team tramite il meccanismo delle *pull request* <sup>2</sup>. Questo processo ha permesso di assicurare che le funzionalità fossero implementate correttamente.

Per assicurare un approccio standardizzato nella scrittura dei messaggi di commit, garantendo una chiara storia di commit e il versionamento automatico, è stato adottato l'approccio *Conventional Commits* <sup>3</sup>. Questo approccio standardizza

<sup>1</sup><https://github.com/>

<sup>2</sup><https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/creating-a-pull-request>

<sup>3</sup><https://www.conventionalcommits.org/en/v1.0.0/>

i messaggi di commit facilitando la comprensione del lavoro svolto e consentendo un controllo più efficace delle modifiche nel repository del progetto.

Un commit è scritto nel seguente formato: `<type>[<ambito opzionale>]: <descrizione>`. Alcuni esempi di tipi di commit sono `feat`, `fix`, `docs` e `style`.

Per imporre lo standard *Conventional Commits*, sono state utilizzate tecniche di commit linting all'interno del repository GitHub del progetto. Questo processo controlla automaticamente i messaggi di commit prima della loro sottomissione, garantendo il rispetto delle convenzioni stabilite.

Nei progetti Kotlin, sono stati utilizzati strumenti aggiuntivi integrati nei pre-commit hooks per migliorare ulteriormente la qualità del codice. Tra questi strumenti:

- **ktlint** per la formattazione automatica del codice secondo le convenzioni stabilite;
- **detekt** per l'analisi statica del codice, rilevando eventuali problemi di qualità e fornendo suggerimenti per migliorare la leggibilità e la manutenibilità del codice.

### 5.1.2 Continuous Integration

La Continuous Integration (*CI*) [22] è una pratica di sviluppo software che coinvolge l'integrazione regolare delle modifiche al codice in un repository condiviso, seguendo procedure automatizzate di compilazione e testing. La sua importanza risiede nella capacità di ottimizzare il flusso di lavoro dello sviluppo, identificando e risolvendo tempestivamente i problemi già nelle fasi iniziali del ciclo di sviluppo. La *CI* garantisce che le modifiche apportate da diversi sviluppatori non causino conflitti o errori, migliorando così la qualità complessiva del codice, riducendo i problemi di integrazione e accelerando la distribuzione del software. Automatizzando tali processi, la *CI* non solo risparmia tempo, ma incoraggia anche la collaborazione e stimola gli sviluppatori a scrivere codice affidabile e manutenibile, garantendo un flusso di sviluppo del software più efficiente e robusto.

Nel progetto, sono state adottate le GitHub Actions <sup>4</sup> come strumento per la Continuous Integration. Le GitHub Actions consentono di eseguire test automatici e analisi del codice ad ogni modifica. Questo approccio garantisce la stabilità del progetto e facilita l'individuazione tempestiva di eventuali errori.

GitHub Actions è un servizio integrato in GitHub che offre agli sviluppatori la possibilità di automatizzare i flussi di lavoro nello sviluppo del software. Si basa sul concetto di workflow, una sequenza di job eseguiti in risposta a eventi specifici. Ad esempio, un workflow può essere attivato quando viene aperta una pull request o quando viene effettuato un commit nel repository. I workflow sono definiti all'interno di un file YAML situato nel percorso `.github/workflows/`. Un esempio di file potrebbe essere: `.github/workflows/main.yml`. Di seguito, nel listing 5.1, è riportato un esempio di workflow che viene attivato quando viene effettuato un commit nel branch principale `main`. È composto da un job chiamato `build` che viene eseguito su una macchina `ubuntu`. Il job consiste in quattro passaggi: prima controlla il codice, poi configura l'ambiente `Python`, quindi installa le dipendenze del progetto e infine esegue uno script personalizzato.

## 5.2 Implementazione sistema os-auto-updates

### 5.2.1 Kotlin Multiplatform

`os-auto-updates` è stato scritto in Kotlin Multiplatform, una tecnologia che, come mostrato in fig. 5.2, è progettata per semplificare lo sviluppo di progetti multipiattaforma <sup>5</sup>. Le piattaforme di destinazione sono:

- **JVM**: permettendo al progetto di essere eseguito su qualsiasi macchina in cui è installata una Java Virtual Machine [23].
- **Nativo**: consentendo al progetto di essere trasferito in codice di basso livello che funziona su Linux e Windows in modo nativo, senza la necessità di una Java Virtual Machine.

---

<sup>4</sup><https://docs.github.com/en/actions>

<sup>5</sup><https://kotlinlang.org/docs/multiplatform.html>

Listing 5.1: Esempio di file per la *CI* su Github.

```
1 name: Custom Workflow
2
3 on:
4   push:
5     branches:
6       - main
7
8 jobs:
9   custom_build:
10    runs-on: ubuntu-latest
11
12    steps:
13    - name: Clone Repository
14      uses: actions/checkout@v2
15
16    - name: Set up Python
17      uses: actions/setup-python@v2
18      with:
19        python-version: '3.9'
20
21    - name: Install Dependencies
22      run: pip install -r requirements.txt
23
24    - name: Run Custom Script
25      run: python custom_script.py
```

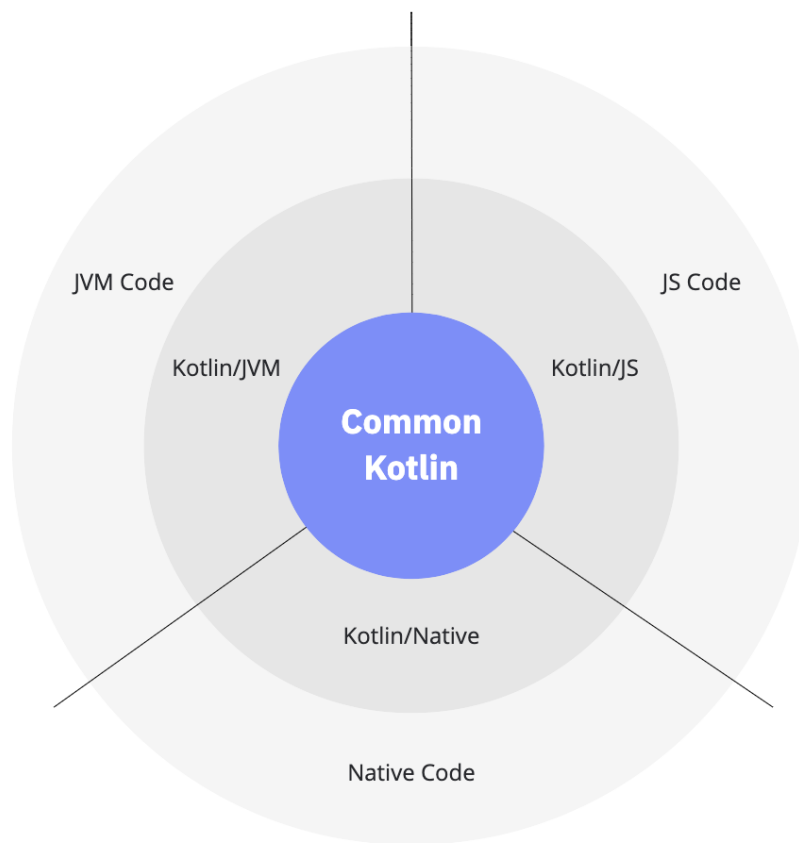


Figura 5.2: Kotlin Multiplatform.

In qualità di progetto Kotlin Multiplatform, una considerevole parte del codice è condivisa tra diverse piattaforme. In alcuni casi è stato, tuttavia, necessario scrivere del codice specifico per ciascuna piattaforma utilizzando il meccanismo *expected/actual* <sup>6</sup>.

A tale proposito, viene presa in esame la classe *OsCommandImpl*, la quale agevola l'esecuzione di comandi nel sistema operativo attraverso la linea di comando. Questo componente riveste un'importanza fondamentale per l'esecuzione di comandi come quelli offerti da Winget. È opportuno sottolineare che l'esecuzione dei comandi su terminale è gestita in modo diverso nell'ambiente della JVM rispetto all'ambiente nativo poiché coinvolgono librerie differenti. Per questo motivo

---

<sup>6</sup><https://kotlinlang.org/docs/multiplatform-expected-actual.html>

Listing 5.2: *Expected OsCommand*.

```
1 package it.unibo.osautoupdates.system.oscommand.impl
2
3 import it.unibo.osautoupdates.system.oscommand.OsCommand
4
5 internal expect class OsCommandImpl(commands: List<String>) : OsCommand
```

Listing 5.3: Implementazione *OsCommand* per JVM.

```
1 package it.unibo.osautoupdates.system.oscommand.impl
2
3 import ...
4
5 /**
6  * JVM implementation of a command to execute in the operating system using the
7  * command line.
8  */
9 internal actual data class OsCommandImpl actual constructor(private val commands:
10 List<String>) : OsCommand(commands) {
11
12     override fun invoke(): Either<OsCommandError, OsCommandOutput> {
13         //JVM version
14     }
15 }
```

è stato adottato il meccanismo *expected/actual*, implementando prima la *expected OsCommandImpl* come mostrato nel listing 5.2, e successivamente sono state implementate le versioni *actual* di *OsCommand* per la JVM (listing 5.3) e la versione *actual* nativa (listing 5.4).

## 5.2.2 Programmazione asincrona

La programmazione asincrona è fondamentale nello sviluppo software moderno, poiché offre la possibilità di migliorare l'efficienza e la reattività delle applicazioni. Questa metodologia consente alle attività di eseguire in modo concorrente senza bloccare il thread di esecuzione principale, permettendo alle applicazioni di gestire più operazioni contemporaneamente. Ciò le rende più scalabili e reattive alle interazioni dell'utente.

La programmazione asincrona in Kotlin si basa sul concetto delle coroutines [24]. Le coroutines possono essere utilizzate come un meccanismo a basso livello per implementare thread leggeri. In altre parole, un singolo thread può ospitare molte coroutines, ma solo una coroutines è in esecuzione in un determinato momento.

Listing 5.4: Implementazione *OsCommand* nativo.

```
1 package it.unibo.osautoupdates.system.oscommand.impl
2
3 import ...
4
5 /**
6  * Native implementation of a command to execute in the operating system using the
7  *   command line.
8  */
9 internal actual data class OsCommandImpl actual constructor(private val commands:
10 List<String>) : OsCommand(commands) {
11
12     override fun invoke(): Either<OsCommandError, OsCommandOutput> {
13         //Native version
14     }
15 }
```

Listing 5.5: Esempio di *suspending function*.

```
1 suspend fun greetAfter(name: String, delayMillis: Long) {
2     delay(delayMillis)
3     println("Hello, $name")
4 }
```

Nel linguaggio Kotlin, le coroutines <sup>7</sup> si basano su due concetti principali: le *suspending computations* e i *coroutine builders*.

Le *suspending computations* rappresentano operazioni che possono essere sospese e riprese in un secondo momento, consentendo al thread di eseguire altre operazioni nel frattempo.

In Kotlin, ci sono due tipi di *suspending computations*:

- **suspending functions**: Sono funzioni che possono mettere in pausa l'esecuzione e restituire il controllo al chiamante fino a quando non è pronta per continuare. Sono contrassegnate dal modificatore *suspend*.
- **suspending lambda**: Sono simili alle *lambda functions* del paradigma funzionale [25], ma possono mettere in pausa la propria esecuzione chiamando altre *suspending functions*. Possono essere utilizzate per eseguire operazioni asincrone all'interno di un blocco di codice.

Le *suspending functions* sono contrassegnate con la parola chiave *suspend* al momento della dichiarazione. Quando una *suspending function* viene chiamata, il

---

<sup>7</sup><https://kotlinlang.org/docs/coroutines-basics.html>

punto nel codice in cui viene invocata la chiamata della funzione rimane implicitamente in attesa fino a quando la funzione non ha completato la sua esecuzione. Questo rende il codice asincrono indistinguibile da quello sincrono, semplificando la comprensione del codice.

C'è, però, una limitazione su come si possono invocare le *suspending functions*: non possono essere chiamate direttamente da funzioni che non siano anch'esse *suspend*. Nell'esempio fornito nel listing 5.5, la funzione `delay()` è una *suspending function*, che sospende l'esecuzione per una determinata durata. Questa funzione è effettivamente invocata da `greetAfter()`, che è anch'essa una *suspending function*.

Per collegare i mondi sincrono e asincrono, si utilizzano i *Coroutine builders*. I *Coroutine builders* sono funzioni classiche non *suspend* che, tramite una *suspending lambda*, creano e avviano le coroutines, permettendo di gestirne il ciclo di vita e il comportamento in modo flessibile. Un esempio che utilizza il *coroutine builder* `runBlocking()` per collegare codice bloccante a codice *suspend* è mostrato nel listing 5.6.

### 5.2.3 Arrow

Nel progetto è stata impiegata la libreria Arrow <sup>8</sup> per avvicinare il codice sviluppato al paradigma funzionale [26]. Arrow mira infatti a promuovere l'approccio alla programmazione funzionale in Kotlin, ispirandosi al lavoro svolto in altri linguaggi, come Scala [27], ma adattando i concetti in modo che siano familiari agli sviluppatori Kotlin.

Si concentra sulla fornitura di strutture dati immutabili, operazioni funzionali e tipi astratti che aiutano a scrivere codice robusto e manutenibile. Di seguito una panoramica di alcuni concetti chiave e funzionalità offerte da Arrow:

- **Strutture dati immutabili:** Arrow fornisce una serie di strutture dati immutabili come *Option*, *Either* e altre. Queste strutture dati favoriscono un approccio funzionale alla gestione dei dati, consentendo operazioni sicure e prevenendo *side effect* indesiderati.

---

<sup>8</sup><https://arrow-kt.io/learn/overview/>

Listing 5.6: Esempio di utilizzo delle coroutines in Kotlin.

```
1 import kotlinx.coroutines.*
2
3 fun main(args: Array<String>) = runBlocking {
4     println("before async call")
5     val result = async {
6         println("inside the async call")
7         delay(1000)
8         println("exiting the async call")
9         100
10    }
11    println("after the async call, before greet")
12    greetDelayed(200)
13    println("after greet trigger")
14    println("${result.await()} ")
15 }
16
17 suspend fun greetDelayed(delayMillis: Long) {
18     delay(delayMillis)
19     println("Hello, World!")
20 }
21
22
23 /*
24 OUTPUT
25 before async call
26 after the async call, before greet
27 inside the async call
28 Hello, World!
29 after greet trigger
30 exiting the async call
31 100
32 */
```

- **Type classes:** Arrow implementa il concetto di type classes, che sono una caratteristica fondamentale della programmazione funzionale. Le type classes consentono la creazione di codice polimorfo, facilitando l'estensione di funzionalità su diversi tipi di dati. Alcuni esempi di type classes in Arrow includono *Functor*, *Monad*, *Traversable*, *Foldable*, e molti altri.
- **Sintassi funzionale:** Arrow offre un insieme di funzioni e operatori per lavorare con le strutture dati immutabili e le type classes. Questa sintassi funzionale include operazioni come *map*, *flatMap*, *traverse*, *fold*, e molti altri.

### 5.2.4 Implementazione PackageURLDeploymentStrategy

Per implementare la *fase* di fetch di una strategia di deployment del software denominata *PackageURLDeploymentStrategy* raffigurata in fig. 4.3, sono stati utilizzati i concetti chiave precedentemente analizzati sulla programmazione funzionale e asincrona in Kotlin.

Questa strategia prende il nome proprio dalla sua fase di *fetch*, la quale si basa sul protocollo HTTP per recuperare un eseguibile da un URL specifico. Durante questa fase, viene utilizzato un client HTTP implementato tramite le funzionalità offerte dalla libreria Ktor <sup>9</sup>, il quale si occupa di scaricare l'eseguibile del software dall'URL specificato.

Nel codice sorgente, come evidenziato nel listato listing 5.7 e listing 5.8, sono state impiegate le funzionalità offerte dalla libreria Arrow in combinazione con le coroutine di Kotlin.

Le coroutine hanno reso possibile il download dell'eseguibile in modo asincrono ed efficiente, mantenendo il codice leggibile e senza bloccare il thread principale. Utilizzando le *suspending functions*, è stato possibile eseguire il download dall'URL specificato senza dover gestire callback annidate o complessi meccanismi di thread.

La libreria Arrow è stata impiegata per gestire i risultati, inclusi gli errori, derivanti dall'operazione di download. Le funzioni *map*, *mapLeft*, e *fold* offerte da Arrow hanno consentito di trasformare e combinare i risultati in modo chiaro e conciso, garantendo una gestione robusta degli errori. Dopo il download dell'eseguibile dall'URL specifico, se l'operazione avviene con successo, il software

---

<sup>9</sup><https://ktor.io/>

sarà mappato nel componente di sistema *ResolvedSoftware*, come rappresentato nel diagramma delle classi in fig. 4.5.

Questa combinazione di coroutine e Arrow ha reso possibile l'implementazione della fase di *fetch* della strategia di deployment in linea con i principi della programmazione funzionale e asincrona.

Listing 5.7: Fase di *fetch* della strategia di deployment *PackageURLDeploymentStrategy*.

```
1 package it.unibo.osautoupdates.deployment.impl
2
3 import arrow.core.EitherNel
4 import arrow.core.raise.either
5 import arrow.core.right
6 import kotlinx.coroutines.runBlocking
7 import ...
8
9 /**
10  * Basic implementation of [DeploymentStrategy].
11  */
12 @Serializable
13 @SerialName(PackageURLDeploymentStrategy.SERIAL_NAME)
14 data class PackageURLDeploymentStrategy(
15     private val packageURL: String,
16     private val packageExtension: String,
17     private val authenticationWith: AuthenticationStrategy? = null,
18     ...
19 ) : DeploymentStrategy {
20
21     /**
22      * Step 0) Fetching the resources needed to install the software.
23      * @param software the [ResolvedSoftware] that is going to be installed.
24      * @return the [ResolvedSoftware] with the resources fetched if the operation
25      *         succeeded,
26      * otherwise [DeploymentError.FetchError]s.
27      */
28 override fun fetchResources(software: ResolvedSoftware): EitherNel<
29     DeploymentError.FetchError, ResolvedSoftware> {
30     val trimmedSwName = software.name.filterNot { it.isWhitespace() } + "-" +
31         software.version
32     val softwareResourceFolder = softwareResourcesDirectoryPath() +
33         PathExtensions.SEPARATOR + trimmedSwName
34     softwareResourceFolder.toPath().mkdir()
35     val artifactName = softwareResourceFolder + PathExtensions.SEPARATOR +
36         trimmedSwName + "." + packageExtension
37     return runBlocking {
38         HttpClient.create().download(packageURL, authenticationWith,
39             artifactName.toPath())
40     }.map {
41         software
42     }
43 }
```

Listing 5.8: Utilizzo di Arrow e coroutine per il download di un'eseguibile software.

```

1 import arrow.core.Either
2 import arrow.core.EitherNel
3 import arrow.core.left
4 import arrow.core.right
5 import arrow.core.toEitherNel
6 import ...
7
8 /**
9  * Extensions for the [HttpClient] class.
10 */
11 object HttpClientExtensions {
12
13     suspend fun HttpClient.download(
14         url: String,
15         authentication: AuthenticationStrategy? = null,
16         path: Path,
17     ): EitherNel<DeploymentError.FetchError, HttpResponse> {
18         return download(url, authentication) { response ->
19             response.byteReadChannelToFile(path)
20         }
21     }
22
23     suspend fun HttpClient.download(
24         url: String,
25         authentication: AuthenticationStrategy? = null,
26         action: suspend (HttpResponse) -> Unit,
27     ): EitherNel<DeploymentError.FetchError, HttpResponse> {
28         logger.info { "Starting download process..." }
29         val clientConfig: HttpRequestBuilder.() -> Unit = {
30             authentication?.configureRequest(this)
31             onDownload { bytesSentTotal, contentLength ->
32                 logger.debug { "Downloaded $bytesSentTotal of $contentLength" }
33             }
34         }
35         return repeatThreeTimes {
36             Either.catchOrThrow<Exception, HttpResponse> {
37                 prepareGet(url, clientConfig).execute { response: HttpResponse ->
38                     action(response)
39                     response
40                 }
41             }.map { response ->
42                 when (response.status) {
43                     in OK..<MultipleChoices -> response.right()
44                     else -> DeploymentError.FetchError(url, response).left()
45                 }
46             }.mapLeft { exception ->
47                 when (exception) {
48                     is IOException -> DeploymentError.FetchError(url, exception)
49                     else -> DeploymentError.FetchError(url, exception)
50                 }
51             }.fold(
52                 ifLeft = { it.left() },
53                 ifRight = { it },
54             ).toEitherNel()
55         }
56     }
57 }

```

---

## Capitolo 6

# Conclusioni e lavori futuri

Il lavoro svolto in questa tesi ha portato alla creazione di un sistema dedicato alla gestione degli aggiornamenti degli applicativi aziendali sul *PC Macchina* dei nuovi centri di lavoro e sezionatrici per la lavorazione del legno prodotti da SCM Group.

Attraverso attività preliminari di studio, è stato deciso di adottare il package manager Winget per la gestione degli applicativi aziendali sul sistema operativo Windows del *PC Macchina*.

Il sistema sviluppato rappresenta un notevole progresso verso una gestione più efficiente degli aggiornamenti software aziendali, includendo funzionalità come il downgrade automatico, l'automazione delle procedure di installazione e la possibilità di effettuare aggiornamenti e rollback remoti del sistema operativo Windows sulle macchine di SCM Group.

Come discusso in precedenza nella sezione 3.2.1, attualmente Winget non offre supporto per la creazione di pacchetti relativi alle applicazioni private sviluppate internamente dall'azienda. È prevista, tuttavia, l'implementazione di questa funzionalità in futuro, rendendo Winget una soluzione più completa e adatta. Di conseguenza, un possibile sviluppo futuro potrebbe includere l'integrazione di queste nuove funzionalità nel sistema, consentendo una gestione più efficiente dei software privati aziendali.

L'implementazione di nuovi componenti in *os-auto-updates* per altri package manager di Windows, come Scoop e Chocolatey, rappresenterebbe certamente un'importante evoluzione per il sistema di gestione degli aggiornamenti. Ciò con-

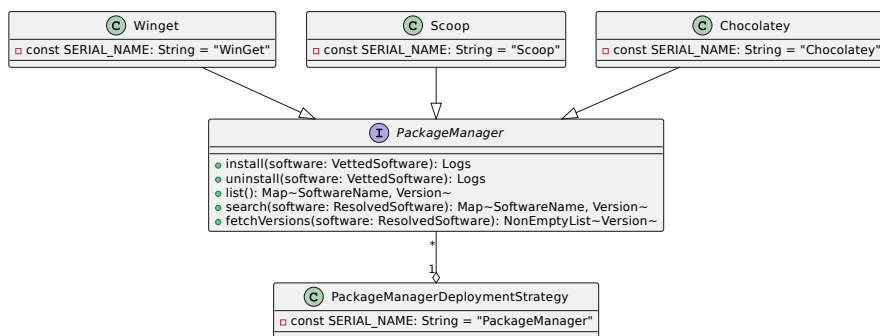


Figura 6.1: Integrazione con Scoop e Chocolatey.

sentirebbe di ampliare la gamma di software gestiti dal sistema e di adottare un approccio più flessibile nella gestione delle applicazioni.

Per realizzare questa integrazione, sarebbe necessario sviluppare nuovi moduli di *os-auto-updates* specifici per ciascun package manager, come illustrato in fig. 6.1, che implementino i metodi di *search*, *list*, *install*, e *uninstall*. Sarebbe, inoltre, importante integrare questi nuovi componenti nella strategia di deployment esistente, chiamata *PackageManagerDeploymentStrategy*, che è stata già analizzata in precedenza. Questo consentirebbe di mantenere coerenza nel sistema e di utilizzare le stesse logiche di gestione degli aggiornamenti indipendentemente dal package manager utilizzato.

Questa integrazione potrebbe rappresentare una sfida tecnica, poiché ogni package manager ha caratteristiche e comportamenti diversi. Tuttavia, una volta implementata con successo, offrirebbe un notevole valore aggiunto al sistema di gestione degli aggiornamenti, migliorandone completezza e adattabilità.

L'applicativo che opera a livello *gateway* deve ancora essere sviluppato. Il suo compito principale sarà quello di individuare gli aggiornamenti disponibili e convertirli in file con formato YAML, che verranno poi forniti in input a *os-auto-updates* in esecuzione su una specifica macchina automatica. Poiché l'applicativo *gateway* fungerà da piattaforma centralizzata per la gestione degli aggiornamenti, sarà essenziale progettarlo tenendo conto degli aspetti legati alla sicurezza. Uno degli aspetti critici riguarderà la sicurezza dei dati e delle comunicazioni durante il trasferimento degli aggiornamenti a *os-auto-updates*. Sarà fondamentale adottare protocolli crittografici robusti per garantire la confidenzialità e l'integrità dei dati

---

trasmessi, proteggendo così le macchine da potenziali attacchi informatici.

Per lo sviluppo dell'applicativo *gateway*, è vantaggioso trarre ispirazione dagli studi condotti sugli *aggiornamenti OTA* nel settore automobilistico. In questo contesto, sono stati sviluppati protocolli dedicati per garantire la sicurezza del firmware durante gli *aggiornamenti over-the-air* [28]. Questi protocolli assicurano l'integrità, l'autenticazione e la riservatezza dei dati durante il processo di aggiornamento, utilizzando le risorse hardware limitate presenti nell'ambiente wireless dei veicoli. Inoltre, sono state esplorate soluzioni basate sulla blockchain per migliorare ulteriormente la sicurezza dei processi di aggiornamento [29]. Sfruttando le caratteristiche della blockchain, come l'immutabilità delle transazioni e la trasparenza del registro distribuito, si mira a garantire un processo di aggiornamento più sicuro, affidabile e trasparente per i veicoli smart. Integrare queste *best practices* nello sviluppo dell'applicativo *gateway* contribuirà complessivamente a rafforzare la sicurezza e l'affidabilità del sistema di gestione degli *aggiornamenti OTA* delle macchine automatiche prodotte da SCM Group.

---

---

# Bibliografia

- [1] K. Zhou, T. Liu, and L. Zhou, “Industry 4.0: Towards future industrial opportunities and challenges,” in *2015 12th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*, 2015, pp. 2147–2152.
- [2] A. Kapse, M. Nimse, A. Bhalekar, U. Zambare, J. Rout, and T. Hinge, “Design and development of special purpose machine for sensor hole punching and bracket pasting operation,” in *2021 International Conference on Computing, Communication and Green Engineering (CCGE)*, 2021, pp. 1–4.
- [3] A. Bonde, N. Mandavgade, and S. Jachak, “Design of multipurpose mechanical machine,” *Materials Today: Proceedings*, vol. 49, pp. 1180–1184, 2022, gC-RASM 2021. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S2214785321045314>
- [4] S. Dick and D. Volmar, “Dll hell: Software dependencies, failure, and the maintenance of microsoft windows,” *IEEE Annals of the History of Computing*, vol. 40, no. 4, pp. 28–51, 2018.
- [5] N. P. DeGuglielmo, S. M. Basnet, and D. E. Dow, “Introduce ladder logic and programmable logic controller (plc),” in *2020 Annual Conference Northeast Section (ASEE-NE)*, 2020, pp. 1–5.
- [6] I. Mugarza, J. L. Flores, and J. L. Montero, “Security issues and software updates management in the industrial internet of things (iiot) era,” *Sensors*, vol. 20, no. 24, 2020. [Online]. Available: <https://www.mdpi.com/1424-8220/20/24/7160>

- [7] J. Bauwens, P. Ruckebusch, S. Giannoulis, I. Moerman, and E. D. Poorter, “Over-the-air software updates in the internet of things: An overview of key principles,” *IEEE Communications Magazine*, vol. 58, no. 2, pp. 35–41, 2020.
- [8] *Introduction to the Internet of Things*, 2018, pp. 1–50.
- [9] D. e. C. H. Kang, Byungseok e Kim, “Internet of everything: un gateway iot autonomo su larga scala,” *IEEE Transactions on Multi-Scale Computing Systems*, vol. 3.
- [10] X. He, S. Alqahtani, R. Gamble, and M. Papa, “Securing over-the-air iot firmware updates using blockchain,” in *Proceedings of the International Conference on Omni-Layer Intelligent Systems*, ser. COINS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 164–171. [Online]. Available: <https://doi.org/10.1145/3312614.3312649>
- [11] E. Blázquez, S. Pastrana, Feal, J. Gamba, P. Kotzias, N. Vallina-Rodriguez, and J. Tapiador, “Trouble over-the-air: An analysis of fota apps in the android ecosystem,” in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 1606–1622.
- [12] S. Halder, A. Ghosal, and M. Conti, “Secure over-the-air software updates in connected vehicles: A survey,” *Computer Networks*, vol. 178, p. 107343, 2020. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1389128619314963>
- [13] H. Guissouma, A. Diewald, and E. Sax, “A generic system for automotive software over the air (sota) updates allowing efficient variant and release management,” in *Information Systems Architecture and Technology: Proceedings of 39th International Conference on Information Systems Architecture and Technology – ISAT 2018*, L. Borzemski, J. Świątek, and Z. Wilimowska, Eds. Cham: Springer International Publishing, 2019, pp. 78–89.
- [14] J. Bannan, 2016.
- [15] R. Siddaway and B. Payette, 2017.

- [16] S. Bokhari, “The linux operating system,” *Computer*, vol. 28, no. 8, pp. 74–79, 1995.
- [17] H. Nava, 2021.
- [18] M. N. Alanazi, “Basic rules to build correct uml diagrams,” in *2009 International Conference on New Trends in Information and Service Science*, 2009, pp. 72–76.
- [19] “Verifica e validazione del software,” in *Conferenza sulle applicazioni tecniche IEEE. Northcon/96. Record della conferenza*.
- [20] “Devops,” *IEEE Software*, vol. 33.
- [21] D. Spinellis, “Git,” *IEEE software*, vol. 29, no. 3, pp. 100–101, 2012.
- [22] M. Meyer, “Continuous integration and its tools,” *IEEE Software*, vol. 31, no. 3, pp. 14–16, 2014.
- [23] “Strumenti didattici per macchine virtuali java,” in *2019 IEEE 15th International Scientific Conference on Informatics*.
- [24] R. Elizarov, M. Beliaev, M. K. Akhin, and I. Usmanov, “Kotlin coroutines: design and implementation,” *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2021. [Online]. Available: <https://api.semanticscholar.org/CorpusID:238992531>
- [25] J. Ramsay, “An introduction to lambda functions and transforms,” in *1963 Antennas and Propagation Society International Symposium*, vol. 1, 1963, pp. 205–211.
- [26] A. Khanfor and Y. Yang, “An overview of practical impacts of functional programming,” in *2017 24th Asia-Pacific Software Engineering Conference Workshops (APSECW)*, 2017, pp. 50–54.
- [27] P. Chiusano, R. Bjarnason, and M. Pilquist, 2023.

- [28] D. K. Nilsson and U. E. Larson, “Secure firmware updates over the air in intelligent vehicles,” in *ICC Workshops - 2008 IEEE International Conference on Communications Workshops*, 2008, pp. 380–384.
- [29] M. Steger, A. Dorri, S. S. Kanhere, K. Römer, R. Jurdak, and M. Karner, “Secure wireless automotive software updates using blockchains: A proof of concept,” in *Advanced Microsystems for Automotive Applications 2017*, C. Zachäus, B. Müller, and G. Meyer, Eds. Cham: Springer International Publishing, 2018, pp. 137–149.