

IDP AI-enabled

Sommario

L'IDP come piattaforma di sicurezza per lo sviluppo AI-assisted	3
L'IDP come "safety net" dello sviluppo AI-assisted	3
Ridurre la dipendenza dalla conoscenza infrastrutturale	4
L'IDP come "paved road" per il codice generato dall'AI	5
Standardizzazione senza rallentare l'innovazione	5
Il ruolo dell'IDP cambia quindi con la diffusione dell'AI	6
Piattaforma Agentica- AI-enabled Internal Developer Platform	7
Agent Builder, Agent Runtime e IDP Guard Rails per il nuovo SDLC	7
1. Executive Summary	7
2. Perché estendere l'IDP agli Agent AI	8
3. Il nuovo modello di SDLC	9
4. Layer Builder – Langflow	10
Tecnologia	10
Obiettivo	10
4.1 Dal prototipo all'artefatto	10
5. Layer Deploy – Kagent	11
Tecnologia	11
6. Perché Kubernetes-native Agent	12
7. Il principio fondamentale: Guard Rails, non Guard Walls	12
Modello 1 – Nessun controllo	12
Modello 2 – Platform troppo restrittiva	13
Modello desiderato	13
8. Le categorie di Guard Rail	13
8.1 Architectural Guard Rails	13
9. Infrastructure Guard Rails	14
10. Technology Guard Rails	14
11. Security Guard Rails	15
Least Agency	15
12. Identity e Secrets	16
13. Versioning degli Agent	16
14. Agent Registry	17

15. Evaluation come nuova fase dello SDLC	17
16. Agent CI/CD	18
17. Golden Path	18
18. Developer Experience	19
19. Il ruolo della Platform Engineering	20
20. Un nuovo contratto tra Developer e Platform	21
21. Maturity Model	21
Level 1 — Experiment	21
Level 2 — Managed Agent	21
Level 3 — Governed Agent	21
Level 4 — Enterprise AgentOps	22
Level 5 — AI Platform	22
22. Messaggio per l'adozione	22
23. Target Architecture	23
24. Conclusione	24

L>IDP come piattaforma di sicurezza per lo sviluppo AI-assisted

L>IDP diventa ancora più importante proprio quando l'AI abbassa la barriera alla produzione di codice.

L'evoluzione dell'AI non riguarda soltanto la nascita degli Agent. Un cambiamento ancora più profondo riguarda il modo in cui viene prodotto il **software tradizionale**. Gli strumenti di AI coding consentono infatti a persone con competenze tecniche inferiori rispetto al passato di generare applicazioni, API, script, microservizi e automazioni funzionanti. Il risultato è un ampliamento significativo della platea di persone in grado di trasformare un'esigenza in codice.

Questo fenomeno rappresenta una grande opportunità, ma modifica anche uno dei presupposti storici del software engineering: **il fatto che chi produce codice possieda necessariamente tutte le competenze necessarie per portarlo in sicurezza in produzione.**

L'AI può infatti rendere molto semplice:

```
Idea
↓
Prompt
↓
Codice generato dall'AI
↓
Applicazione funzionante
```

ma il percorso:

```
Applicazione funzionante
↓
Applicazione sicura
↓
Applicazione manutenibile
↓
Applicazione osservabile
↓
Applicazione conforme
↓
Applicazione operabile in produzione
```

rimane un problema di **software engineering, platform engineering e operations**.

È proprio in questo spazio che l>IDP assume un'importanza strategica.

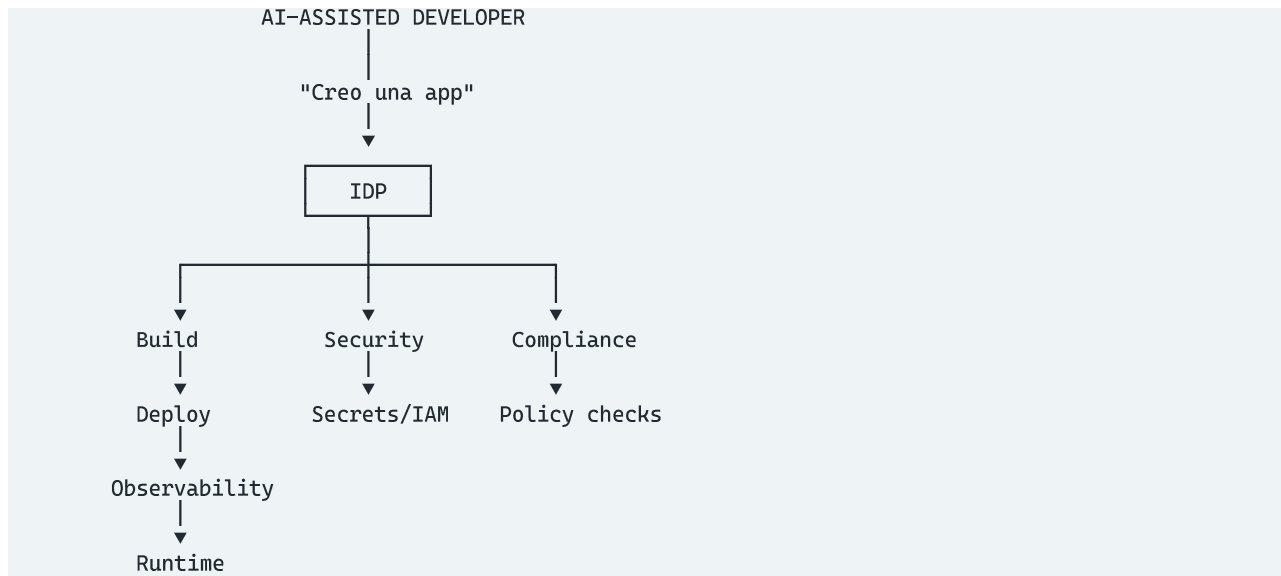
L>IDP come "safety net" dello sviluppo AI-assisted

L>IDP permette di separare due responsabilità che, con l'AI, diventano sempre più importanti da distinguere:

la capacità di creare software e la capacità di gestire software in produzione.

Un developer, un data analyst, un product specialist o un altro profilo abilitato dall'AI può essere messo nelle condizioni di creare un'applicazione senza dover necessariamente conoscere in profondità Kubernetes, networking, IAM, container security, observability, storage o CI/CD.

La Platform assorbe questa complessità attraverso **Golden Path, template, policy e servizi self-service**:



Il principio diventa quindi:

L'AI democratizza la produzione del codice; l'IDP democratizza la capacità di eseguire quel codice in modo governato.

Ridurre la dipendenza dalla conoscenza infrastrutturale

Senza una Platform, l'aumento della capacità di generare codice può produrre un effetto collaterale: la crescita del numero di applicazioni che devono essere successivamente prese in carico dai team tecnici.

Ogni applicazione può infatti introdurre autonomamente:

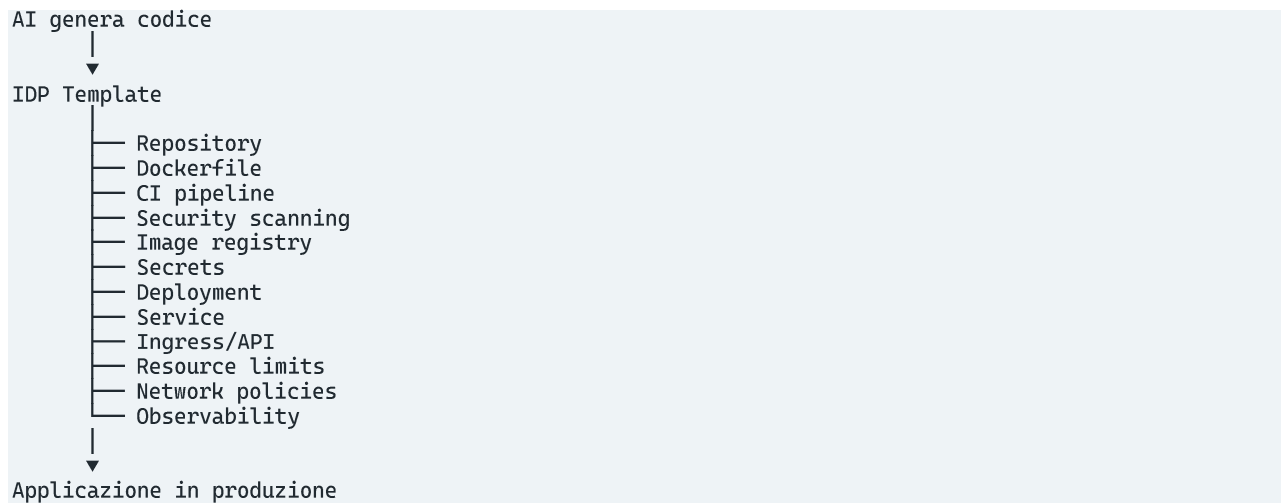
- un framework;
- una modalità di deployment;
- un database;
- un sistema di logging;
- un meccanismo di autenticazione;
- una gestione dei secret;
- un container;
- una configurazione di rete;
- una pipeline;
- una modalità di monitoring.

L'IDP introduce invece un **percorso standardizzato**.

Per esempio, l'utente potrebbe chiedere all'AI:

"Crea una REST API in Python con PostgreSQL."

L'AI può produrre il codice, mentre la Platform può fornire automaticamente il contesto necessario per trasformarlo in un workload aziendale:



L'utente non deve necessariamente conoscere l'implementazione di ogni singolo componente.

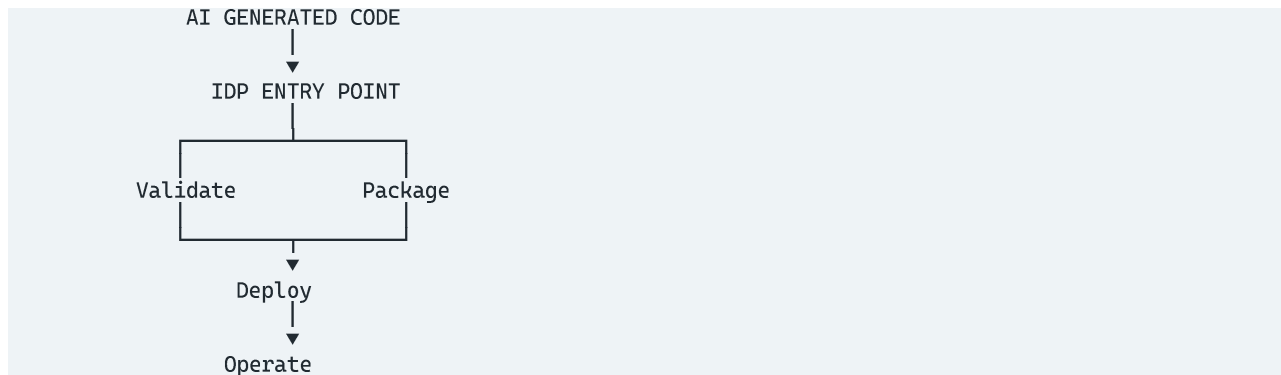
Deve invece conoscere **il contratto della Platform** e le modalità con cui richiedere i servizi.

L'IDP come "paved road" per il codice generato dall'AI

Questo porta a un concetto particolarmente importante nel nuovo SDLC:

AI-generated code should enter production through a paved road.

La Platform definisce una strada preconfigurata e supportata:



In questo modello l'IDP non deve impedire la sperimentazione.

Al contrario, deve permettere di sperimentare rapidamente, separando però:

Experimentation

da

Production readiness.

Un'applicazione può quindi essere creata rapidamente con l'aiuto dell'AI, ma per raggiungere un ambiente produttivo deve attraversare i controlli e i servizi messi a disposizione dalla Platform.

Standardizzazione senza rallentare l'innovazione

Un ulteriore vantaggio è che la standardizzazione può essere applicata **a livello di Platform e non necessariamente a livello di codice.**

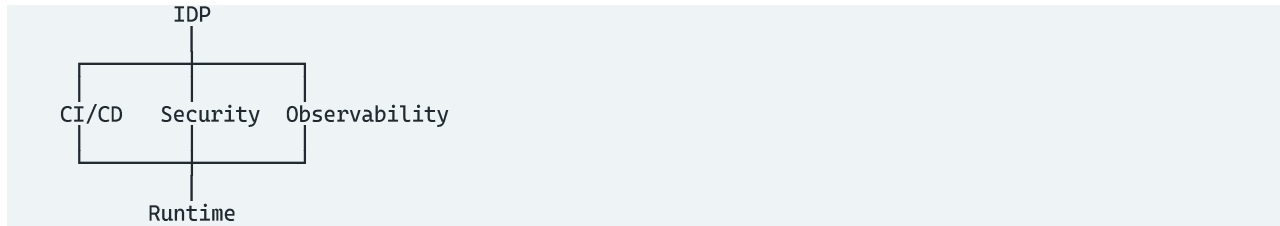
Due team possono utilizzare linguaggi o framework diversi:

```

Team A → Python / FastAPI
Team B → Go
Team C → Node.js
Team D → Java

```

ma utilizzare lo stesso contratto operativo:



Questo consente di preservare la libertà tecnologica dove è utile, concentrando la standardizzazione sugli aspetti che devono essere governati centralmente.

Il ruolo dell'IDP cambia quindi con la diffusione dell'AI

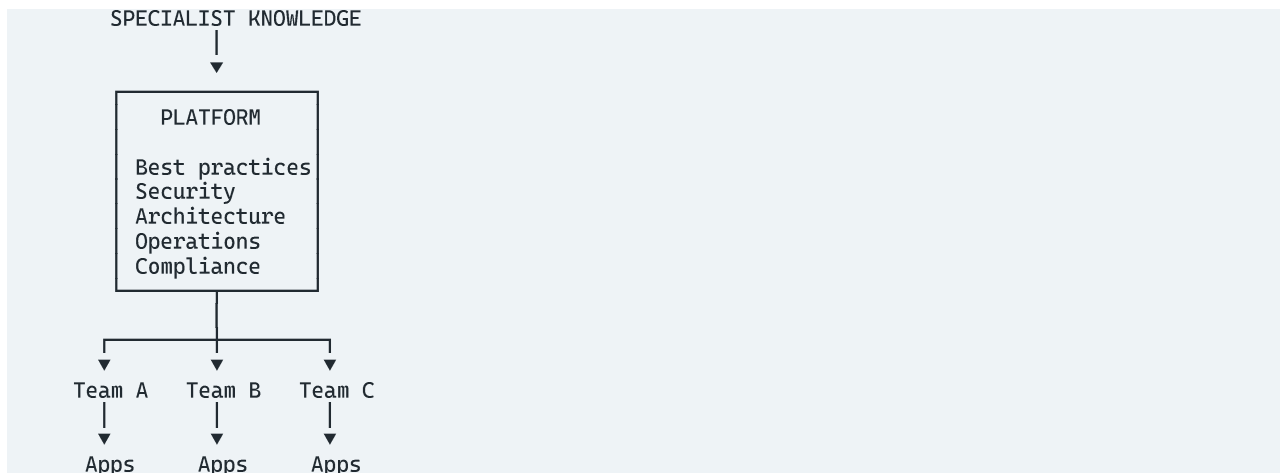
Nel modello tradizionale l'IDP aumenta la produttività di sviluppatori già tecnicamente esperti.

Nel modello AI-assisted assume una funzione più ampia:

L'IDP diventa il meccanismo che consente di estendere la capacità di sviluppo software a una popolazione più ampia senza estendere nella stessa proporzione la complessità operativa.

Questo non significa che la competenza tecnica diventi meno importante. Al contrario, aumenta il valore dei team che progettano la Platform, le architetture, i guard rail e i Golden Path.

La conoscenza specialistica viene infatti **spostata dal singolo progetto verso la Platform**, dove può essere codificata una volta e riutilizzata da molti team.



Il risultato è un modello nel quale **l'AI aumenta la capacità di creare software, mentre l'IDP aumenta la capacità dell'organizzazione di governare quel software.**

Ed è proprio questa combinazione — **AI per democratizzare la software creation + IDP per industrializzarne l'esecuzione** — che rende la Platform un elemento centrale del futuro SDLC, anche indipendentemente dalla presenza dei suoi specifici componenti agentici.

Piattaforma Agentica- AI-enabled Internal Developer Platform

La Platform non si limita a fornire strumenti per costruire Agent AI: fornisce un percorso governato per portarli dall'idea al runtime, mantenendo sicurezza, compliance, versioning, osservabilità e standard aziendali.

Agent Builder, Agent Runtime e IDP Guard Rails per il nuovo SDLC

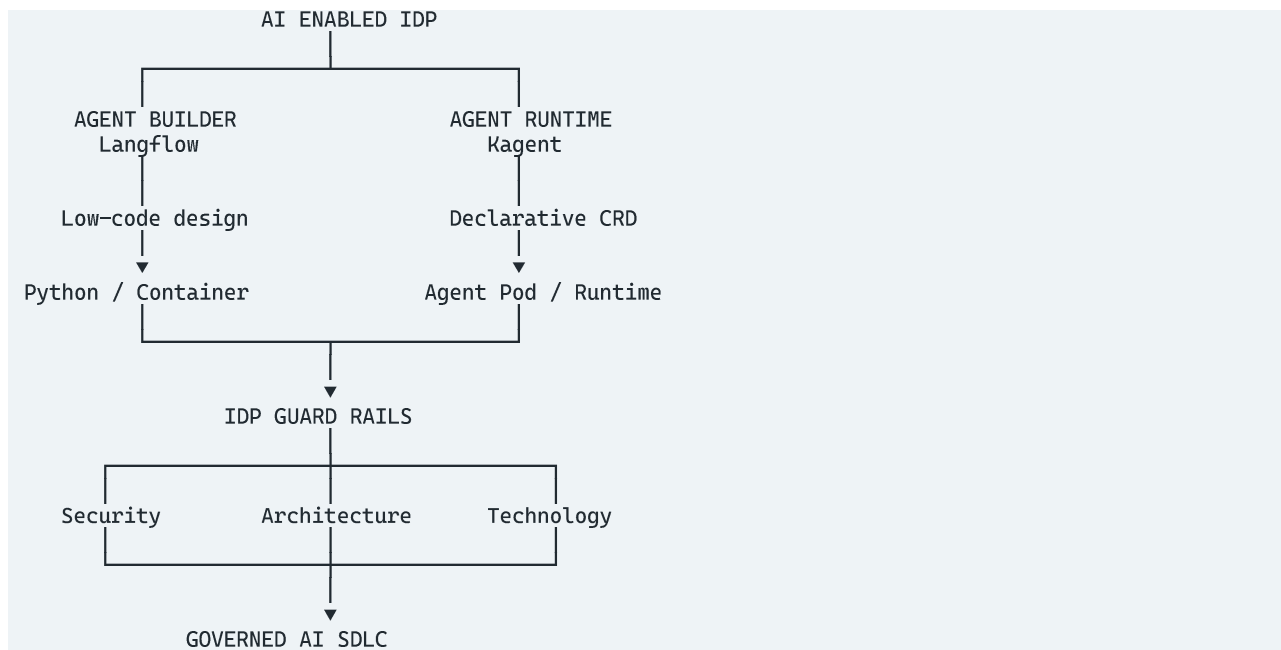
1. Executive Summary

Il modello tradizionale di Software Development Lifecycle sta evolvendo da un processo in cui il developer scrive prevalentemente codice a un modello nel quale il software viene sempre più spesso **progettato, generato, verificato e gestito attraverso sistemi AI agentici**.

In questo nuovo scenario un'Internal Developer Platform (IDP) non dovrebbe limitarsi a fornire Kubernetes, CI/CD, registry e servizi infrastrutturali.

La Platform deve diventare il **sistema di riferimento attraverso cui anche gli AI Agent vengono costruiti, verificati, distribuiti e governati**.

L'architettura proposta introduce due capability principali:

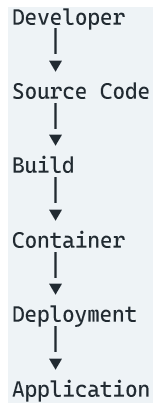


L'obiettivo non è quindi fornire semplicemente "un tool per creare agent", ma creare un **percorso standardizzato e governato per trasformare un'idea AI in un servizio operativo aziendale**.

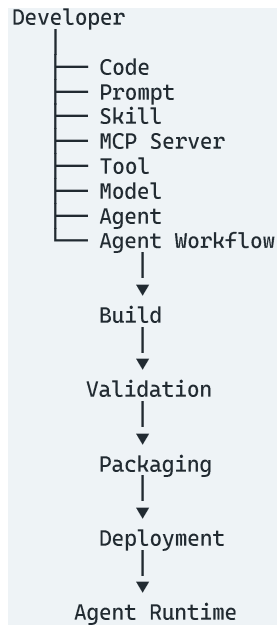
2. Perché estendere l'IDP agli Agent AI

L'adozione degli Agent AI introduce un nuovo insieme di artefatti software.

Nel modello tradizionale:



Nel modello AI-enabled:



L'Agent diventa quindi un nuovo tipo di **software artifact**.

Questo comporta una conseguenza importante:

Gli Agent devono essere trattati con gli stessi principi di engineering applicati al software tradizionale.

Devono quindi essere:

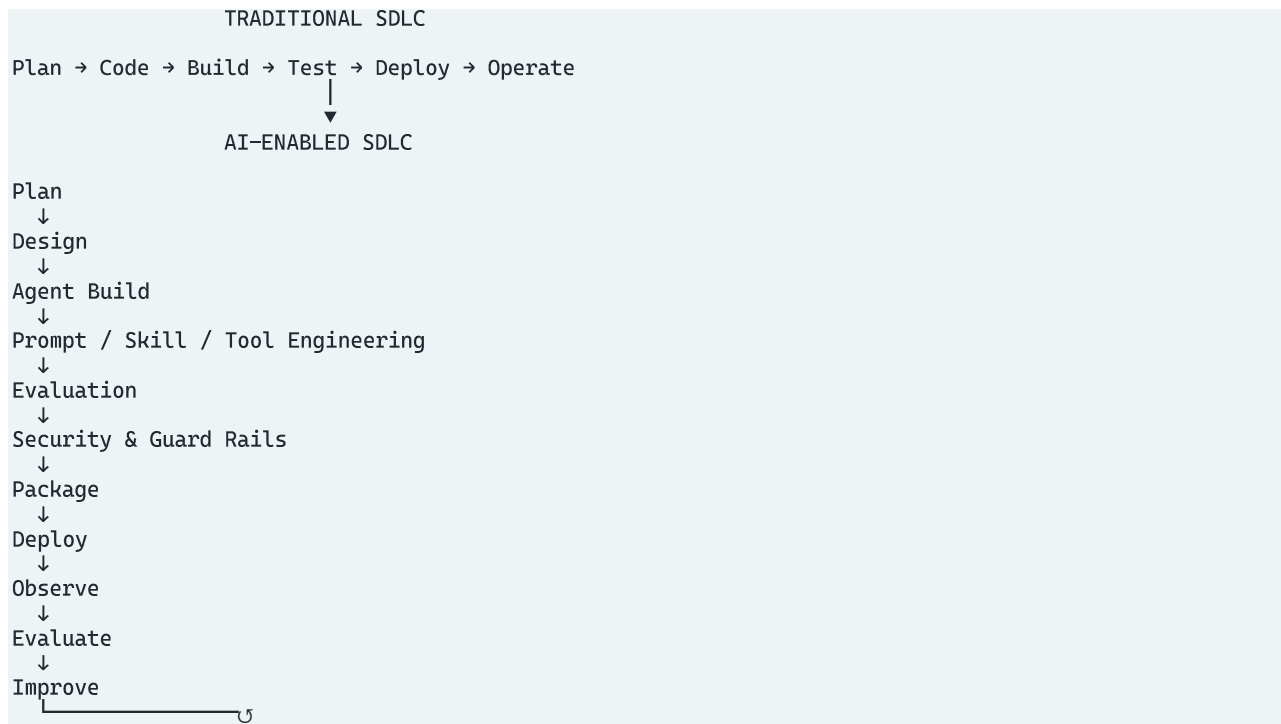
- versionati;
- riproducibili;
- testabili;
- osservabili;
- distribuiti attraverso pipeline;
- sottoposti a controlli di sicurezza;
- associati a ownership;

- governati secondo policy aziendali;
- aggiornabili e ritirabili.

3. Il nuovo modello di SDLC

L'AI introduce una nuova dimensione nello SDLC.

Possiamo rappresentarlo come:



La differenza fondamentale è il **feedback loop**.

Un Agent non è necessariamente un artefatto statico.

Può evolvere attraverso:

- modifica del prompt;
- modifica delle istruzioni;
- modifica dei tool;
- modifica del modello;
- modifica della knowledge base;
- modifica delle policy;
- modifica del workflow.

Per questo l>IDP deve supportare un ciclo:

Build → Evaluate → Deploy → Observe → Improve

4. Layer Builder – Langflow

Il primo layer della piattaforma è il **Builder Layer**.

Tecnologia

Langflow

Obiettivo

Fornire agli sviluppatori un ambiente low-code per progettare:

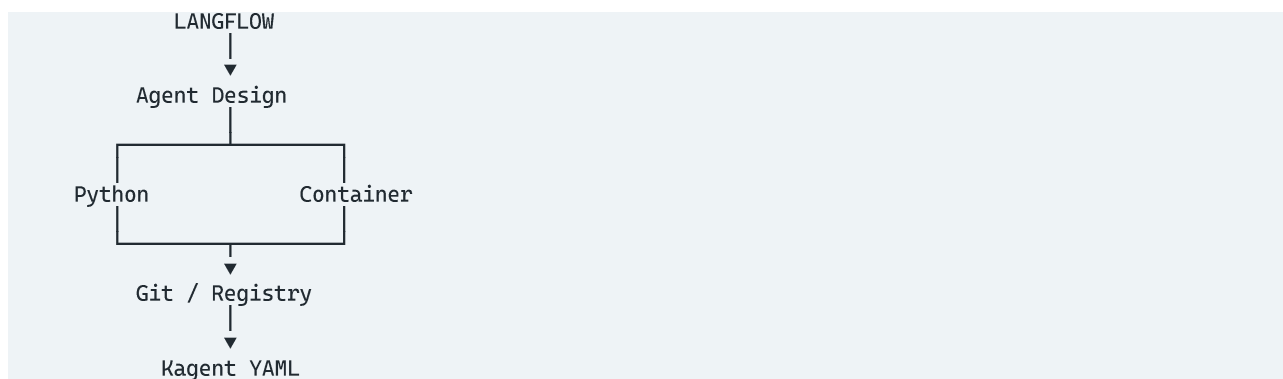
- agent;
- workflow;
- prompt;
- model interaction;
- tool;
- componenti;
- integrazioni;
- RAG;
- MCP;
- logiche di orchestrazione.

L'obiettivo è abbassare la barriera di ingresso alla costruzione di sistemi AI senza eliminare i principi di software engineering.

4.1 Dal prototipo all'artefatto

Langflow non deve essere considerato il runtime aziendale dell'Agent.

Il suo ruolo principale nella Platform è:



L'output del Builder diventa quindi un **artefatto versionabile**.

Questo è fondamentale per evitare il modello:

Developer → crea Agent → clicca Deploy

sostituendolo con:

```
Developer
↓
Build
↓
Version
↓
Validate
↓
Review
↓
Deploy
```

5. Layer Deploy – Kagent

Il secondo layer è il **Agent Runtime / Deployment Layer**.

Tecnologia

Kagent

Il ruolo di Kagent è fornire un modello **Kubernetes-native e declarative** per rappresentare gli Agent.

L'Agent diventa una risorsa dichiarativa:

```
apiVersion: ...
kind: Agent
metadata:
  name: my-agent
spec:
  ...
```

Il modello diventa quindi:

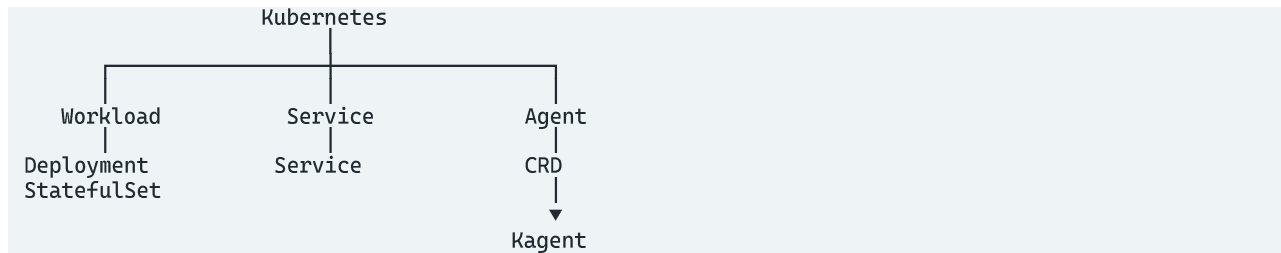
```
Git
↓
Agent YAML
↓
GitOps
↓
Kagent
↓
Agent
↓
Pod
```

Questo porta gli Agent dentro il modello operativo già adottato dalla Platform.

6. Perché Kubernetes-native Agent

L'utilizzo di CRD per rappresentare gli Agent produce un vantaggio architetturale importante.

L'Agent diventa una risorsa gestibile attraverso gli stessi meccanismi utilizzati per le applicazioni:



Questo consente di applicare agli Agent:

- RBAC;
- Namespace isolation;
- NetworkPolicy;
- ResourceQuota;
- LimitRange;
- secrets management;
- observability;
- admission control;
- policy enforcement;
- GitOps;
- audit;
- lifecycle management.

L'Agent non diventa quindi un'entità "speciale" fuori dalla Platform.

Diventa un **first-class workload della Platform**.

7. Il principio fondamentale: Guard Rails, non Guard Walls

La Platform deve evitare due estremi.

Modello 1 – Nessun controllo

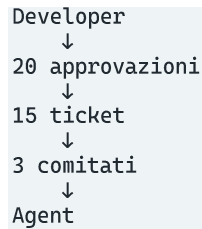


Questo porta rapidamente a:

- agent non versionati;
- credenziali embedded;
- tool non autorizzati;
- modelli non approvati;

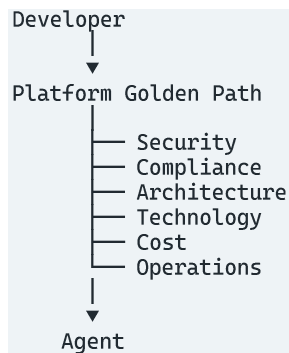
- immagini non certificate;
- vulnerabilità;
- data leakage;
- costi incontrollati;
- difficoltà di audit.

Modello 2 – Platform troppo restrittiva



Questo porta all'abbandono della Platform.

Modello desiderato



Quindi:

Guard Rails devono rendere facile fare la cosa corretta, non semplicemente impedire di fare la cosa sbagliata.

8. Le categorie di Guard Rail

Propongo quattro categorie principali.

8.1 Architectural Guard Rails

Definiscono **come deve essere costruito un Agent**.

Esempi:

- pattern architetturali approvati;
- separazione Builder/Runtime;
- utilizzo di MCP per tool integration;
- obbligo di externalizzazione dei secret;
- separazione tra agent e knowledge store;
- standard per RAG;

- pattern di comunicazione tra Agent;
- timeout;
- retry;
- circuit breaker;
- dependency management.

9. Infrastructure Guard Rails

Definiscono **dove e con quali risorse può girare l'Agent**.

Esempi:

```
CPU / Memory limits
├── ResourceQuota
├── LimitRange
└── Agent profile
```

Network:

```
Agent
├── allowed APIs
├── allowed MCP
├── allowed databases
└── allowed external endpoints
```

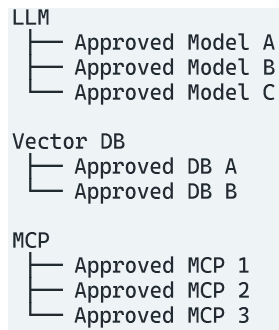
Security:

- Pod Security Standards;
- NetworkPolicy;
- non-root;
- read-only filesystem quando possibile;
- image signing;
- image scanning;
- SBOM;
- secret management;
- workload identity;
- namespace isolation.

10. Technology Guard Rails

La Platform deve anche definire **quali tecnologie possono essere utilizzate**.

Ad esempio:



Questo evita che ogni team costruisca una propria AI platform.

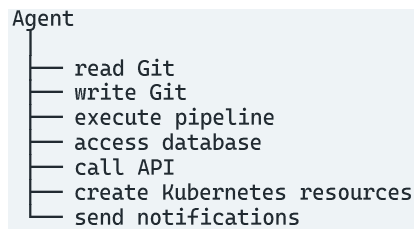
La Platform diventa quindi un **Technology Marketplace governato**.

11. Security Guard Rails

Questa è probabilmente la categoria più importante.

Un Agent può avere capacità operative molto superiori a quelle di un'applicazione tradizionale.

Per esempio:



Il problema non è soltanto:

"L'Agent è sicuro?"

ma:

"Quali capacità gli sono state concesse?"

Questo porta al principio:

Least Agency

Non solo:

Least Privilege

ma:

Least Agency

ovvero concedere all'Agent **solo le capacità necessarie per svolgere il proprio compito**.

12. Identity e Secrets

Ogni Agent dovrebbe avere una propria identity.

Non:

```
Agent
↓
shared-api-key
```

ma:

```
Agent
↓
Workload Identity
↓
Policy
↓
Resource
```

I secret devono essere gestiti dalla Platform e non contenuti:

```
✗ Git
✗ YAML
✗ Dockerfile
✗ Prompt
✗ Python source
```

ma attraverso il secret management aziendale.

13. Versioning degli Agent

Uno dei cambiamenti più importanti rispetto al software tradizionale è che **non deve essere versionato soltanto il codice**.

Il vero Agent artifact può essere:

```
Agent
├── source code
├── prompt
├── system instructions
├── model
├── model parameters
├── tools
├── MCP servers
├── knowledge sources
├── dependencies
├── container
├── Kagent manifest
└── evaluation dataset
```

Quindi il versioning deve comprendere l'intero **Agent Package**.

Un possibile modello:

```
agent-customer-support
├── agent.yaml
├── prompts/
├── src/
├── evaluations/
├── Dockerfile
├── requirements.txt
└── README.md
```

14. Agent Registry

L'IDP dovrebbe progressivamente introdurre un **Agent Registry / Catalog**.

Il catalogo potrebbe contenere:

```
Agent
├── Owner
├── Version
├── Status
├── Description
├── Model
├── Tools
├── MCP dependencies
├── Data classification
├── Security profile
├── Evaluation results
├── Runtime
└── Lifecycle
```

Questo permette di passare da:

"Abbiamo tanti Agent"

a:

"Abbiamo un inventario governato degli Agent aziendali."

15. Evaluation come nuova fase dello SDLC

Nel software tradizionale:

```
Build
↓
Unit Test
↓
Integration Test
↓
Deploy
```

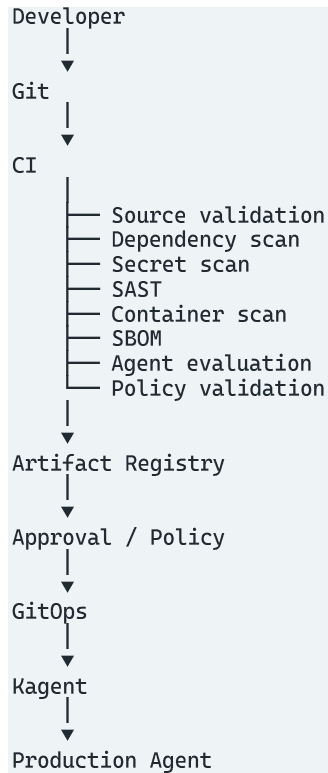
Per gli Agent:

```
Build
↓
Functional Test
↓
Security Test
↓
Prompt Evaluation
↓
Tool Evaluation
↓
Hallucination / Grounding Evaluation
↓
Policy Evaluation
↓
Deploy
```

L'evaluation diventa quindi una **quality gate della Platform**.

16. Agent CI/CD

L'IDP dovrebbe introdurre una pipeline standard:



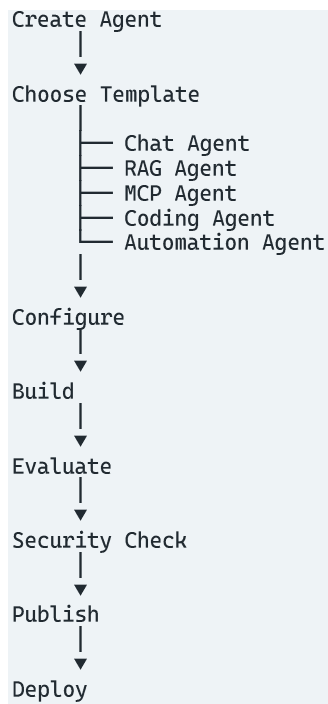
Questo è il punto in cui **AgentOps** entra naturalmente nel DevOps esistente.

17. Golden Path

Per favorire l'adozione non bisogna chiedere agli sviluppatori di conoscere tutti i controlli.

La Platform deve fornire un **Golden Path**.

Esempio:



Il developer dovrebbe percepire:

"Creo un Agent come creo un'applicazione."

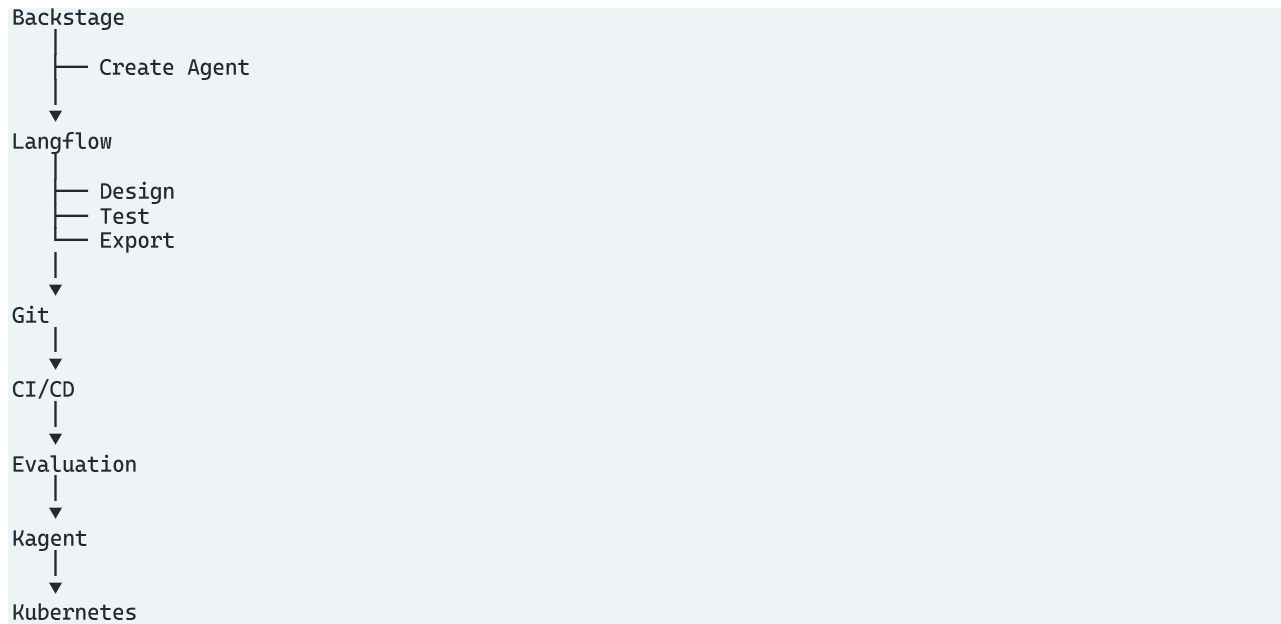
Non:

"Devo imparare 15 tecnologie AI."

18. Developer Experience

L'adozione dell'IDP dipenderà molto dalla Developer Experience.

Il percorso ideale dovrebbe essere:



Backstage può diventare il **portale di accesso alla capability**, mentre Langflow rimane il Builder e Kagent il Runtime.

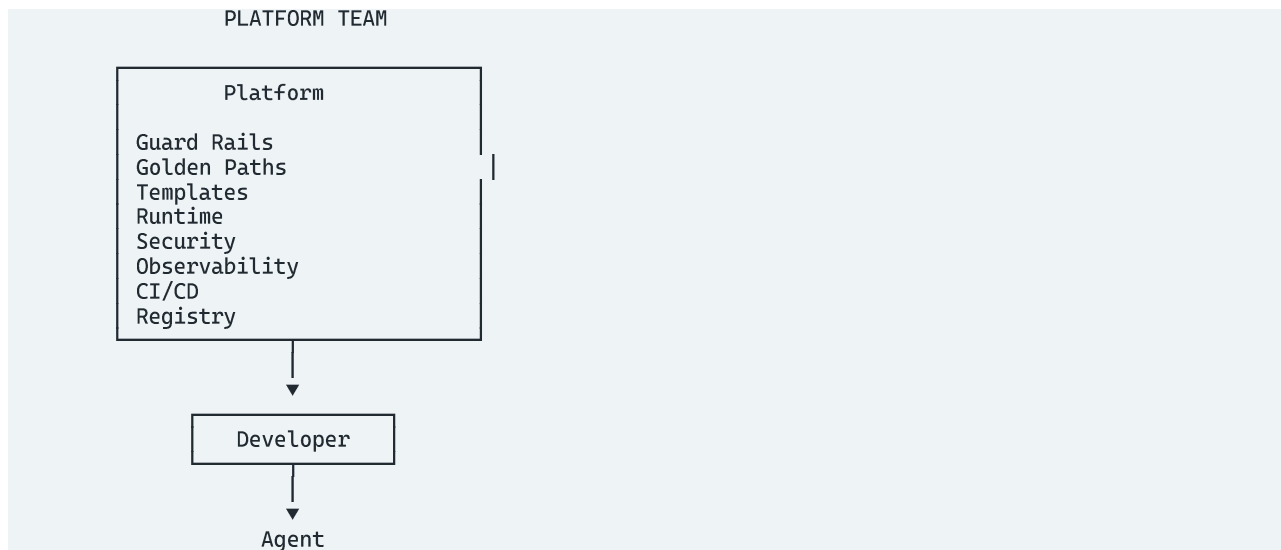
19. Il ruolo della Platform Engineering

Questo introduce una nuova responsabilità per il team Platform.

La Platform non deve costruire gli Agent al posto dei developer.

Deve costruire **il sistema nel quale gli Agent possono essere costruiti in modo sicuro**.

La distinzione è fondamentale:



Il risultato è **platform as a product** anche per l'AI.

20. Un nuovo contratto tra Developer e Platform

L'IDP può formalizzare un **Agent Contract**.

Il developer dichiara:

```
What does my Agent do?
What data does it access?
Which tools does it use?
Which model does it use?
Who owns it?
What risk profile does it have?
```

La Platform garantisce:

```
Where can it run?
Which identities can it use?
Which resources can it access?
Which models are allowed?
Which tools are allowed?
How is it monitored?
How is it audited?
How is it deployed?
```

Questo crea una separazione molto chiara tra:

Business responsibility

e

Platform responsibility.

21. Maturity Model

L'adozione può essere gestita progressivamente.

Level 1 — Experiment

```
Developer
↓
Langflow
↓
Local Agent
```

Nessun deployment enterprise.

Level 2 — Managed Agent

```
Langflow
↓
Git
↓
Container
↓
Kagent
```

Versioning e deployment controllato.

Level 3 — Governed Agent

Aggiungiamo:

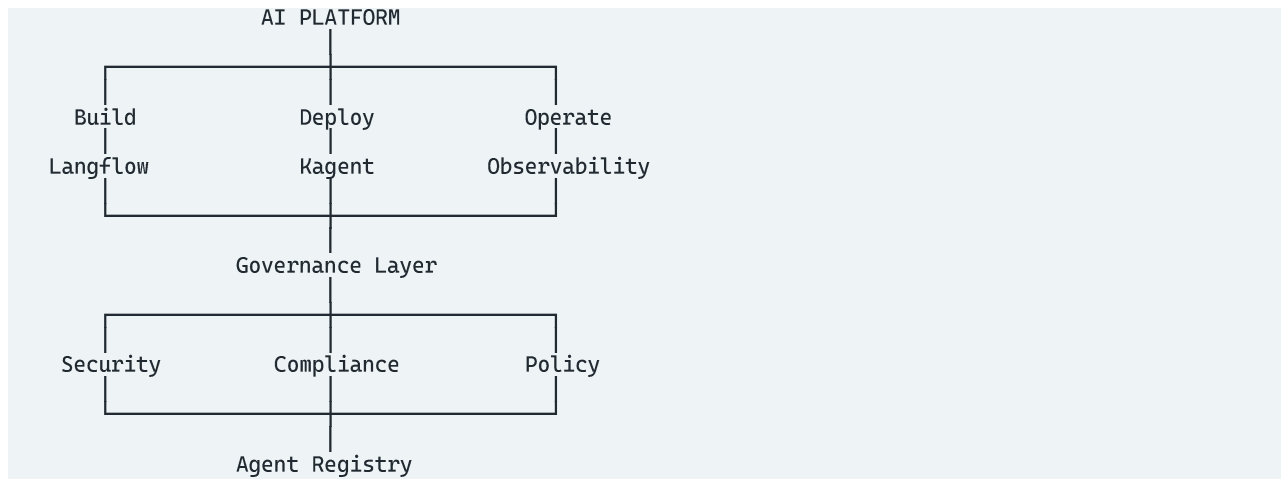
Security
Evaluation
Policy
Observability
RBAC
Secrets

Level 4 — Enterprise AgentOps

Agent Registry
+
Automated Evaluation
+
Policy as Code
+
GitOps
+
Observability
+
Lifecycle Management

Level 5 — AI Platform

La Platform diventa un vero **AI Operating Model**:



22. Messaggio per l'adozione

Il messaggio da portare ai developer non dovrebbe essere:

"La Platform controlla come costruisci il tuo Agent."

Ma:

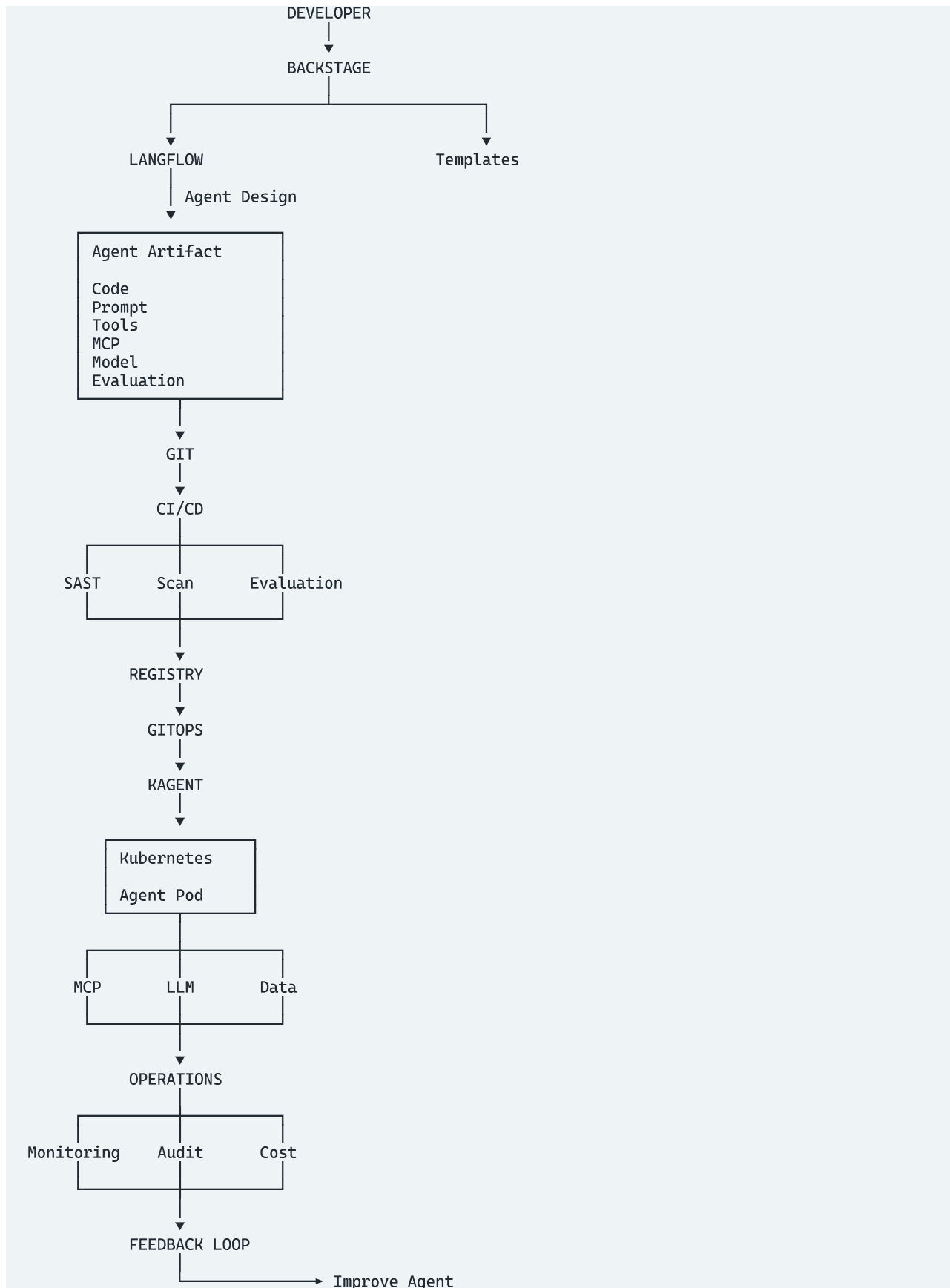
"La Platform ti fornisce tutto ciò che ti serve per costruire e portare in produzione un Agent senza dover reinventare ogni volta sicurezza, infrastruttura, deployment, osservabilità e compliance."

Questa distinzione è fondamentale per l'adozione.

Il valore della Platform non è limitare la creatività dello sviluppatore.

È trasformare la complessità enterprise in capability riutilizzabili.

23. Target Architecture



24. Conclusione

L'introduzione di Langflow e Kagent nell'IDP non dovrebbe essere considerata semplicemente l'aggiunta di due nuovi strumenti tecnologici.

Rappresenta l'evoluzione della Platform da:

| **Internal Developer Platform**

a:

| **AI-enabled Internal Developer Platform**

dove gli Agent diventano una nuova categoria di software artifact.

Il valore principale della Platform diventa quindi la capacità di fornire contemporaneamente:

Freedom to build

*

Guard rails to operate

*

Governance to scale

Il developer mantiene la libertà di progettare il comportamento dell'Agent, mentre la Platform garantisce che il percorso verso la produzione sia coerente con gli standard aziendali.

In questo modello **security, compliance, versioning, evaluation, observability e lifecycle management non vengono aggiunti dopo la costruzione dell'Agent**: diventano proprietà intrinseche del percorso di delivery.

Questo è il passaggio fondamentale verso un modello di SDLC nel quale **codice, AI model, prompt, tool, MCP, workflow e Agent vengono trattati come componenti governati dello stesso sistema software**.