Comment nous avons construit un Copilot d'orchestration Workday en langage naturel

Une expérience de livraison Workday pilotée par le langage naturel

Gabriel Aubert Desjardins Tyler Miezitis

2026-04-24

Table des matières

This article is also available in English — Read the English version.

Vue d'ensemble

Tout a commencé comme un problème d'intégration Workday ordinaire — mais l'expérience a rapidement dérivé vers une autre question : pouvait-on concevoir et livrer une orchestration complète principalement en langage naturel ?

L'objectif n'était pas de construire quelque chose de parfait. Il s'agissait de voir jusqu'où on pouvait pousser cette approche dans un scénario réel : synchroniser une Time Off Service Date depuis les données du worker vers un custom object, de façon sûre et avec suffisamment d'observabilité pour diagnostiquer les échecs.

Ce qui a rendu l'expérience intéressante, c'est la méthode de livraison. Nous avons pu concevoir, valider, construire, déployer et déboguer l'orchestration principalement en langage naturel, en utilisant un ensemble de skills IA spécialisés plutôt qu'un assistant générique.

En pratique, cela ressemblait à recréer un Workday Orchestration Copilot.

> **i** Nouveau dans Workday Orchestration ?
>
> Workday Orchestration est un outil d'intégration low-code intégré à la plateforme Workday qui permet de concevoir et d'automatiser des processus métier multi-étapes via un éditeur visuel de flux. Si vous n'êtes pas familier avec cet outil, l'aperçu des intégrations Workday est un bon point de départ avant de poursuivre la lecture.

Pourquoi cette idée est devenue d'actualité

Le déclencheur n'était pas un enthousiasme abstrait pour l'IA. C'était un changement concret sur la plateforme.

Quand Workday a rendu le Workday Developer CLI généralement disponible, cela a changé la forme pratique du workflow de livraison. Selon l'annonce officielle, WDCLI est devenu l'interface en ligne de commande unifiée pour construire, tester, valider et déployer les applications Workday Integration et Extend sur l'ensemble du cycle de vie applicatif. L'annonce soulignait également des capacités clés : uploader des apps vers App Hub, les télécharger, les déployer sur des tenants, les promouvoir entre environnements, et s'authentifier de façon programmatique via un modèle de system user sécurisé.

C'était important parce que cela créait une surface de livraison crédible centrée sur le terminal, pour les builders Workday. Une fois que Workday lui-même avait rendu le cycle de vie CLI réel, la question logique suivante était simple : si la plateforme peut désormais être pilotée depuis le terminal, une pile d'agents en langage naturel peut-elle devenir la couche opérationnelle qui pilote ce cycle de vie ?

C'est à ce moment que le projet a cessé d'être un simple exercice d'orchestration et a commencé à ressembler à une expérience de construction d'un équivalent de Workday Orchestration Copilot.

Sources :

— Annonce Workday Community : Product Update : Workday Developer CLI now Generally Available
— Démo produit : Workday CLI GA Demo on Vimeo

La pile de skills

Le workflow reposait sur un ensemble restreint de skills ciblés. Chacun avait la responsabilité exclusive d'une partie du problème, et c'est cette séparation qui a rendu le système fiable.

wdcli

Ce skill est l'opérateur de livraison. Il connaît le cycle de vie du Workday Developer CLI, l'ordre des commandes, la différence entre les commandes sûres en lecture seule et les commandes qui modifient l'état, et les vérifications pratiques nécessaires après un upload ou un déploiement.

Dans ce projet, cela importait parce que la sortie WDCLI n'était pas toujours fiable en elle-même. Nous avons observé des cas où `wdcli app upload` affichait à la fois un message de succès et un message d'échec tout en créant quand même une nouvelle version dans App Hub. Le skill a donc évolué de « connaît les commandes » à « traite l'état d'App Hub comme la source de vérité ». C'est le type d'apprentissage opérationnel qu'un assistant générique ne possède généralement pas.

wd-wql

Ce skill est la couche de découverte et de conception de requêtes. Il explique comment penser WQL comme un service REST versionné, comment découvrir les data sources et les champs depuis les métadonnées plutôt que de deviner des alias, et quand WQL est le bon outil par rapport à REST ou SOAP.

C'était essentiel parce que l'orchestration ne pouvait pas être conçue de façon sûre avant de savoir ce que le tenant exposait réellement. Une grande partie du risque dans la livraison Workday vient des fausses hypothèses sur les alias, les data sources et la disponibilité des champs. `wd-wql` réduit ce risque en poussant le workflow vers un raisonnement fondé d'abord sur les métadonnées.

wd-wql-executor

Ce skill est le pont entre la conception et la réalité. Il transforme un statement WQL théorique en un test live sur le tenant. En d'autres termes, il ferme la boucle entre « cette requête semble correcte » et « cette requête retourne réellement les lignes dont nous avons besoin ».

Cette distinction compte plus qu'il n'y paraît. Dans de nombreux projets d'intégration, les équipes s'arrêtent à la syntaxe. Dans ce workflow, le skill executor nous a forcés à valider que la requête fonctionnait réellement dans le tenant et que la forme du résultat était alignée avec la logique de l'orchestration.

wd-api-test-runner

Ce skill valide la plomberie API. C'est le moyen le plus rapide de distinguer les problèmes de credentials, les échecs OAuth, les problèmes de disponibilité du service et les erreurs de chemin des problèmes de conception de l'orchestration.

Cette séparation était critique dans cette histoire. Avant de mettre en cause l'orchestration, nous avons utilisé les vérifications au niveau API pour confirmer si le chemin de service, l'authentification et le comportement de l'endpoint étaient cohérents. Cela a évité un mode d'échec courant dans les projets Workday : déboguer la mauvaise couche.

wd-orchestration

Ce skill maîtrise le langage de conception d'orchestration lui-même. Il comprend les types de nœuds, les patterns `_type` et `_value`, la syntaxe des expressions, le scope upstream, le comportement des boucles, la gestion des erreurs, les integration messages, et les frameworks d'orchestration réutilisables comme OSK.

C'est donc lui qui a été le moteur de raisonnement central du projet. Il a été utilisé pour structurer le flux principal, organiser la suborchestration, améliorer les messages de runtime, et diagnostiquer le 404 à l'exécution alors que le build lui-même semblait avoir réussi.

Pourquoi la séparation des skills est déterminante

Cette séparation s'est avérée fondamentale en pratique.

— La conception d'une requête n'est pas le même problème que son exécution
— La santé de l'API n'est pas le même problème que le JSON de l'orchestration
— Le packaging de l'app n'est pas le même problème que le diagnostic en production

La pile de skills a fonctionné parce que chaque skill avait un contrat étroit. Ensemble, ils ont formé une chaîne d'outils agents pratique pour la livraison Workday, sans prétendre qu'une seule invite pouvait comprendre chaque couche avec la même précision.

Du besoin métier à une conception fonctionnelle

La première étape n'était pas la génération de code. C'était la découverte du tenant.

En utilisant les skills orientés WQL, nous avons confirmé quelles data sources et quels alias existaient réellement dans le tenant, comment la Time Off Service Date pouvait être lue, et à quoi ressemblait le champ du custom object du point de vue de la récupération. Cela importait parce que les intégrations Workday échouent souvent quand les équipes supposent que le tenant correspond aux exemples génériques.

Une fois le contrat de données clarifié, le skill d'orchestration a aidé à traduire le besoin en un flux concret :

— lire les lignes de workers via WQL
— boucler sur chaque résultat
— comparer les valeurs source et cible
— appeler une suborchestration uniquement quand une mise à jour est requise
— émettre des integration messages pour la traçabilité

Ce n'était pas une simple génération de JSON. Cela nécessitait de comprendre les chemins d'exécution Workday, les règles de scope, les frontières de sous-flux et le comportement des nœuds API.

Validation API avant de faire confiance à l'orchestration

Avant de faire confiance à l'orchestration, nous avons validé la direction API directement.

En utilisant les outils de test API, nous avons confirmé que le service REST du custom object était le bon chemin de mise à jour, que la forme du payload était valide, et que le service attendait `worker.id` plutôt que `employeeID`. Cette séparation était importante : elle nous a permis de distinguer un problème d'orchestration d'un problème de contrat de service.

En d'autres termes, le projet a progressé parce que nous n'avons pas deviné. Nous avons validé chaque couche indépendamment.

WDCLI comme couche de livraison

Une fois l'orchestration en place, `wdcli` est devenu le mécanisme de livraison.

Cela a introduit ses propres apprentissages :

- la vraie racine de l'app devait être identifiée correctement
- les dossiers de documentation du projet n'étaient pas suffisants ; le code source de l'app Workday devait contenir `appManifest.json`
- `wdcli app upload` pouvait produire une sortie terminal contradictoire
- la vraie source de vérité était l'état de la version dans App Hub et l'état du build, pas le message terminal seul
- `wdcli app deploy` pouvait échouer avec une exception côté CLI tout en laissant une ambiguïté sur l'état réel de la plateforme

Cela a changé le workflow. Au lieu de supposer que la sortie CLI était authoritative, nous avons commencé à vérifier chaque upload via `wdcli app versions` et chaque build via `wdcli app builds`.

Cela a aussi révélé une conséquence architecturale utile du positionnement de WDCLI par Workday lui-même. L'annonce officielle cadrait explicitement WDCLI comme l'interface pour automatiser les pipelines de build et de test, et pour gérer le cycle de vie complet des apps via des workflows en ligne de commande. Une fois que c'était devenu réel, il était logique de concevoir la couche IA environnante autour de ces mêmes étapes :

- découvrir
- valider
- construire
- uploader
- déployer
- inspecter
- déboguer

C'est l'une des raisons pour lesquelles cette histoire est plus qu'une expérience de prompt one-off. Elle s'aligne sur la surface de livraison que Workday a lui-même choisi d'exposer.

Le moment où l'histoire est devenue intéressante

L'orchestration a compilé avec succès. L'upload a réussi. Le build a passé.

Puis le comportement en production a raconté une autre histoire.

Les integration messages indiquaient initialement que les enregistrements étaient mis à jour, mais quand nous avons vérifié les données cibles dans Workday, le champ restait vide. C'est là que le skill d'orchestration et l'approche de débogage sont devenus critiques.

Nous avons amélioré les integration messages pour qu'ils ne rapportent plus un succès de façon inconditionnelle. À la place, ils ont commencé à logger :

— l'identifiant du worker
— la valeur TOSD source
— le code de statut HTTP de la réponse
— le corps brut de la réponse API

Cette seule amélioration a changé complètement le processus de débogage.

La cause racine du 404

L'exécution suivante a exposé le vrai problème immédiatement.

L'API retournait un `404` avec une réponse indiquant que l'alias du tenant avait été traité comme une partie du chemin de la ressource. Le problème n'était pas la référence du worker, ni le payload. Le problème était le chemin REST lui-même.

Le nom du tenant avait été codé en dur dans le chemin API du custom object :

```
/customObject/v2/<tenant>/customObjects/timeOffServiceDate?updateIfExists=true
```

À l'intérieur d'un appel d'orchestration Workday en contexte de tenant, c'était incorrect. L'orchestration s'exécutait déjà dans le contexte du tenant, donc le nom du tenant ne devait pas apparaître dans le chemin REST.

Le correctif consistait à supprimer le segment tenant et à utiliser le chemin de service relatif au tenant :

```
/customObject/v2/customObjects/timeOffServiceDate?updateIfExists=true
```

C'était un exemple fort de pourquoi le succès du build ne suffit pas. L'orchestration a compilé parfaitement et a quand même échoué à l'exécution parce que le chemin de la requête était sémantiquement incorrect.

C'était aussi un exemple fort de pourquoi les skills spécialisés surpassent la résumé générique. Le skill d'orchestration savait qu'il fallait améliorer le logging autour de `responseStatusCode` et `response.toString()`. Une fois ces détails visibles dans l'integration event, l'erreur de runtime a cessé d'être mystérieuse.

L'orchestration finale

Après la découverte du tenant, la validation API, le débogage itératif et le correctif du chemin REST, voici l'orchestration qui tourne en production.

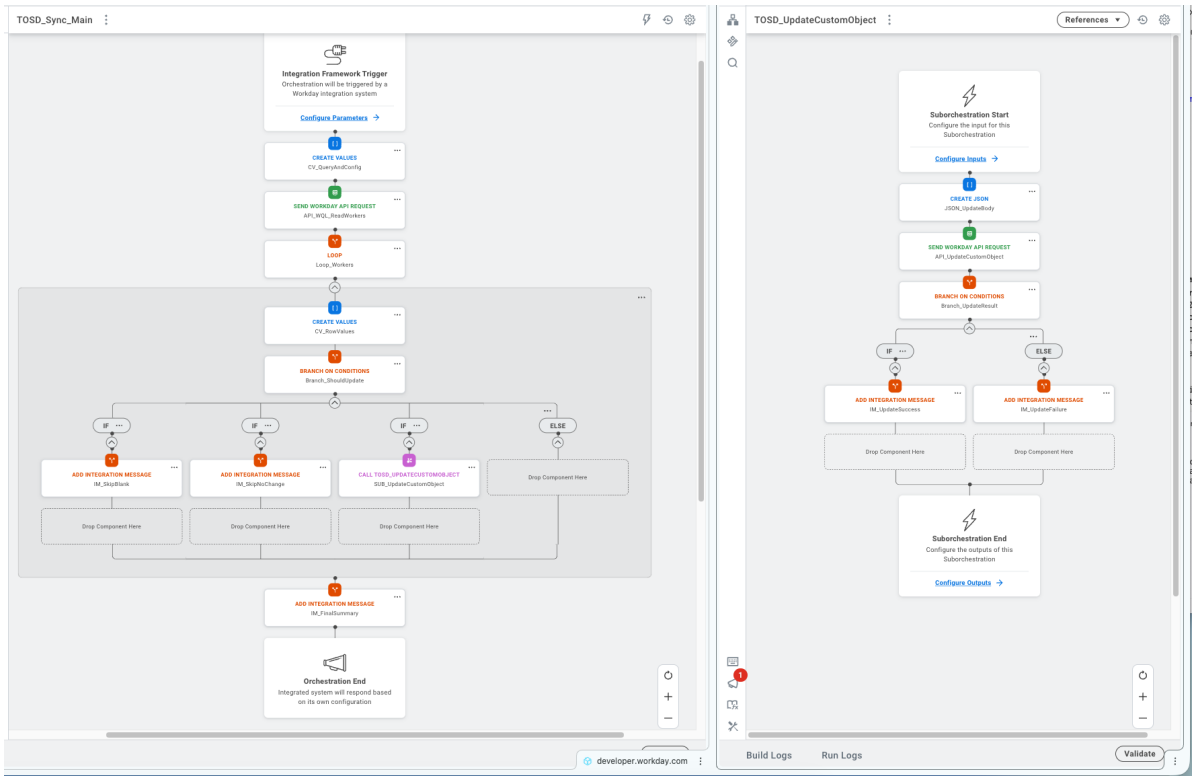

FIGURE 1 : L'orchestration TOSD_Sync_Main complétée (gauche) et sa suborchestration TOSD_UpdateCustomObject (droite), construites entièrement par coordination en langage naturel avec des skills Workday spécialisés.

Le flux principal lit les workers via WQL, boucle sur chaque ligne, compare les valeurs source et cible, et délègue les mises à jour à la suborchestration uniquement quand nécessaire. La suborchestration gère l'appel REST, branche selon la réponse, et émet des integration messages structurés dans les deux cas — succès et échec.

Ce que le package montre

Le code d'orchestration, le snapshot de la racine de l'app, et les cinq skills utilisés dans ce projet sont disponibles sur demande. L'enjeu n'est pas simplement de dire « l'IA a aidé ». Il s'agit de montrer comment le workflow peut être lu, challengé et reproduit par un autre ingénieur.

Pourquoi cela ressemble à une histoire de Copilot

Ce qui vaut la peine d'être documenté ici, ce n'est pas seulement le correctif lui-même. C'est le processus.

Nous avons utilisé le langage naturel pour coordonner :

— la découverte des data sources
— la conception des requêtes WQL
— l'exécution WQL
— la validation API
— la conception de l'orchestration
— le débogage en production
— le packaging et le déploiement via WDCLI

C'est effectivement un workflow de copilot spécialisé pour Workday Extend.

Au lieu de demander à un modèle générique d'improviser sur chaque couche, nous avons utilisé des skills spécialisés portant chacun une connaissance opérationnelle d'une partie de la plateforme. Le résultat : un raisonnement plus fiable, un diagnostic plus rapide, et moins de fausses hypothèses.

La vue d'ensemble

Ce que cette expérience indique, c'est un pattern, pas un cas isolé.

Le langage naturel devient beaucoup plus précieux quand il est connecté à des skills spécialisés, des outils vérifiables, et des workflows tenant-aware. Dans ce cadre, l'IA ne génère pas seulement du texte. Elle aide à réaliser un vrai travail de livraison — conception, validation, déploiement, débogage.

Le timing compte aussi. La disponibilité générale de WDCLI a créé la surface pratique qui permet à ce type d'expérience d'avoir de l'importance. Une fois que le cycle de vie de la plateforme peut être piloté de façon crédible depuis le terminal, il devient raisonnable de placer une couche d'orchestration en langage naturel par-dessus.

C'est pourquoi cette histoire est plus grande qu'une seule orchestration. Elle montre comment une chaîne d'outils basée sur des agents peut fonctionner comme un Workday Orchestration Copilot pratique pour les builders utilisant Codex, Claude, Gemini, OpenCode ou des environnements agents similaires.

Nous continuerons à explorer cet espace — on a l'impression de n'avoir fait qu'effleurer la surface de ce que orchestration et langage naturel pourraient devenir dans Workday.

Note de transparence — IA générative

Des outils d'IA générative ont été utilisés pour l'organisation des sources, la révision du texte, l'aide à la traduction et la vérification de la mise en forme. Les auteurs conservent l'entière responsabilité du choix des sources, de l'interprétation des résultats, de la validation des références et du contenu final. Les sorties IA ont été relues, éditées et vérifiées par les auteurs. Aucun outil d'IA n'est crédité à titre d'auteur.