NetSuite SuiteScript : Client vs User Event vs Scheduled

By houseblend.io Publié le 14 avril 2026 37 min de lecture

Résumé analytique

La plateforme SuiteScript de NetSuite propose plusieurs types de scripts – notamment les **Client Scripts**, les **User Event Scripts** et les **Scheduled Scripts** – chacun étant conçu pour des contextes et des cas d'utilisation spécifiques. Les Client Scripts s'exécutent dans le navigateur de l'utilisateur sur les formulaires d'enregistrement, **réagissant aux événements de l'interface utilisateur** (modifications de champs, modifications de sous-listes, enregistrement de fiches, etc.) (Source: docs.oracle.com). Les User Event Scripts s'exécutent sur le serveur NetSuite **lors des événements du cycle de vie d'un enregistrement** (création, chargement, soumission, modification, suppression) (Source: docs.oracle.com), permettant une validation et une automatisation personnalisées au moment de l'enregistrement ou du chargement. Les Scheduled Scripts s'exécutent également sur le serveur, mais **de manière asynchrone selon un calendrier temporel ou via des déclencheurs explicites**, gérant les tâches par lots et le traitement en arrière-plan (Source: docs.oracle.com) (Source: www.brokenrubik.com). Ces différences fondamentales – contexte d'exécution (navigateur vs serveur), mécanisme de déclenchement (événements UI vs événements d'enregistrement vs temps), caractéristiques de performance et limites de gouvernance (généralement 1 000 unités/exécution pour les scripts client et user event vs 10 000 pour les scripts planifiés (Source: docs.oracle.com) (Source: docs.oracle.com) – déterminent quand et comment chaque type de script est utilisé.

Ce rapport fournit une comparaison approfondie des Client, User Event et Scheduled Scripts dans SuiteScript 2.x. Nous examinons leur **architecture, leurs déclencheurs et leur utilisation typique**, en citant la documentation officielle d'Oracle et des sources expertes, et nous présentons des exemples de référence. Nous analysons les considérations de performance, les limites de gouvernance et les meilleures pratiques issues des guides d'Oracle (Source: docs.oracle.com) (Source: docs.oracle.com) et des blogs spécialisés (Source: www.tvarana.com) (Source: www.brokenrubik.com). Des études de cas illustrent comment les organisations exploitent chaque type de script (pour la validation de l'interface utilisateur, l'application côté serveur, les tâches par lots, etc.). Enfin, nous discutons des orientations futures de SuiteScript (notamment SuiteScript 2.1/TypeScript et l'évolution de l'interface utilisateur de NetSuite) et de leurs implications pour le choix du type de script approprié.

Introduction

NetSuite est un système de Planification des Ressources d'Entreprise (ERP) de premier plan basé sur le cloud. Sa plateforme **SuiteCloud** permet une personnalisation approfondie via SuiteScript (une API basée sur JavaScript), permettant aux entreprises d'adapter NetSuite à des exigences complexes. SuiteScript a évolué au fil du temps : *SuiteScript 1.0* (Classic) a introduit les types de scripts de base à la fin des années 2000, tandis que *SuiteScript 2.x* (à partir de 2015 environ) a modernisé l'API avec des modules de style AMD et a introduit de nouveaux types de scripts (par exemple, Map/Reduce, scripts d'installation SDF) (Source: docs.oracle.com). L'accent actuel (SuiteScript 2.x, incluant le support de 2.1 et TypeScript) encourage le code modulaire et intègre des fonctionnalités telles que le SuiteCloud Development Framework (SDF) et TypeScript (Source: blogs.oracle.com).

Au sein de SuiteScript, les **types de scripts** correspondent au moment et à la manière dont le code s'exécute. Les trois types principaux abordés ici sont :

- **Client Scripts** : S'exécutent dans le *navigateur client*. Ils s'exécutent immédiatement en réponse aux actions de l'interface utilisateur sur les formulaires (chargement de page, changement de champ, insertion de ligne, enregistrement de fiche, etc.) (Source: docs.oracle.com). Ils sont utilisés pour la *logique côté client en temps réel* (par exemple, validation des entrées, comportements dynamiques des champs).
- **User Event Scripts** : S'exécutent sur le *serveur NetSuite* en réponse aux événements d'enregistrement (avant chargement, avant soumission, après soumission) (Source: docs.oracle.com). Ils gèrent les tâches côté serveur telles que l'application de règles métier lors de l'enregistrement, le remplissage d'enregistrements associés ou le déclenchement d'actions de suivi.
- **Scheduled Scripts** : S'exécutent sur le *serveur* de manière asynchrone selon un calendrier ou à la demande (Source: docs.oracle.com). Ils gèrent le traitement en arrière-plan – par exemple, la synchronisation nocturne des données, les mises à jour en masse ou la génération de rapports – sans interaction de l'utilisateur.

Choisir le bon type de script est crucial pour la performance et l'exactitude. Les Client Scripts peuvent fournir un retour immédiat aux utilisateurs (Source: docs.oracle.com), mais sont limités au contexte du navigateur et peuvent s'exécuter plus lentement sur des machines peu performantes (Source: suiterep.com). Les User Event Scripts peuvent appliquer de manière fiable des règles sur le serveur (Source: docs.oracle.com), mais l'exécution d'un trop grand nombre de scripts ou de scripts trop longs peut ralentir le traitement des enregistrements (Source: docs.oracle.com) (Source: www.tvarana.com). Les Scheduled Scripts gèrent les tâches lourdes hors ligne (Source: docs.oracle.com), mais ils consomment des unités de gouvernance importantes et ne peuvent pas interagir directement avec l'interface utilisateur.

Ce rapport examine chaque type de script en détail, compare leurs capacités et contraintes, et fournit des directives (avec un ton académique et des citations) sur le moment où chacun est approprié. Nous intégrons également le *contexte historique* (versions de SuiteScript), les meilleures pratiques actuelles et les tendances futures du scripting NetSuite.

Contexte et évolution de SuiteScript

SuiteScript est né au sein de la plateforme SuiteCloud de NetSuite pour permettre l'ajout de code personnalisé au-delà des configurations standard. SuiteScript 1.0 (l'API originale) utilisait une bibliothèque de fonctions `nlapi*` et des types de scripts appelés « types d'événements » (par exemple, `pageInit`, `saveRecord`). Dans SuiteScript 2.x (introduit vers 2015), Oracle est passé à une architecture modulaire (style AMD/RequireJS) et a standardisé l'environnement de script (Source: docs.oracle.com). Les changements clés dans la version 2.x incluent :

- **Points d'entrée vs Types d'événements** : Dans SuiteScript 2.x, le concept de « points d'entrée » remplace les « types d'événements » de la version 1.0. Chaque type de script définit des fonctions de point d'entrée (par exemple, `pageInit`, `fieldChanged` pour les scripts client ; `beforeLoad`, `beforeSubmit`, `afterSubmit` pour les événements utilisateur ; `execute` pour les scripts planifiés) qui reflètent les déclencheurs de la version 1.0 (Source: docs.oracle.com).
- **Nouveaux types de scripts** : SuiteScript 2.x a ajouté le script **Map/Reduce** (pour le traitement de données volumineuses en parallèle) et les **scripts d'installation SDF** (pour les tâches automatisées de déploiement de SuiteApp) (Source: docs.oracle.com). Ceux-ci n'existaient pas dans la version 1.0.
- **Limites de gouvernance et améliorations** : Chaque type de script a des unités de gouvernance définies par exécution (Source: docs.oracle.com) (Source: docs.oracle.com). SuiteScript 2.x a également introduit une meilleure gestion asynchrone (par exemple, les rendements et la replanification de SuiteScript 2.1, le rendement de Map/Reduce, bien que les scripts planifiés manquent toujours de points de rendement) (Source: docs.oracle.com) (Source: docs.oracle.com). SuiteScript 2.1 (vers la fin des années 2020) a encore amélioré l'expérience des développeurs en ajoutant le **support des modules ES, des typages TypeScript plus riches** et des préférences de rétrocompatibilité (Source: blogs.oracle.com) (Source: www.brokenrubik.com). Le SuiteCloud Development Framework (SDF) encourage l'utilisation de fichiers sources et le contrôle de version pour les scripts (Source: docs.oracle.com) (Source: blogs.oracle.com). En pratique, les développeurs

SuiteScript devraient privilégier la dernière version 2.x (2.1 en 2026) pour les nouveaux travaux, en utilisant la documentation officielle pour les points d'entrée et les API.

La documentation d'Oracle souligne que SuiteScript 2.x *conserve tous les types de scripts de la version 1.0* et en ajoute de nouveaux (Source: docs.oracle.com). Par exemple, les **scripts client, user event et planifiés** sont disponibles à la fois en 1.0 et en 2.x (avec quelques changements dans les paramètres et de nouvelles fonctionnalités) (Source: docs.oracle.com) (Source: docs.oracle.com). Cette continuité permet aux comptes construits sur l'ancien SuiteScript de passer à la version 2.x assez facilement (par exemple, `nlapiSetRecoverPoint` et `nlapiYieldScript` sont obsolètes dans les scripts planifiés 2.x car le modèle d'exécution 2.x rend leur besoin inutile (Source: docs.oracle.com). Comprendre cet historique est important : de nombreux scripts hérités peuvent encore exister, mais les nouveaux développements doivent utiliser les normes SuiteScript 2.x pour la cohérence et le support.

Aperçu des types de scripts SuiteScript

En plus des trois types principaux (Client, User Event, Scheduled), NetSuite propose d'autres types de SuiteScript : Suitelets (pages UI), RESTlets (API), Map/Reduce (traitement par lots), Portlets (composants de tableau de bord), Mass Update, Workflow Action Scripts, etc. Chacun a des points d'entrée et des cas d'utilisation distincts. Le tableau 1 (ci-dessous) résume les caractéristiques clés des Client, User Event et Scheduled Scripts. Les sections ultérieures détailleront chaque ligne. En citant les documents Oracle, toutes les informations sont tirées des guides officiels SuiteScript (Source: docs.oracle.com) (Source: docs.oracle.com) (Source: docs.oracle.com).

ASPECT	CLIENT SCRIPT	USER EVENT SCRIPT	SCHEDULED SCRIPT
Contexte d'exécution	Dans le navigateur (UI client) (Source: docs.oracle.com)	Sur le serveur NetSuite (backend) (Source: docs.oracle.com)	Sur le serveur NetSuite (arrière-plan) (Source: docs.oracle.com)
Événements déclencheurs	Événements UI : pageLoad (mode édition), fieldChange, lineInit, validateLine, saveRecord, etc (Source: docs.oracle.com)	Événements d'enregistrement : beforeLoad, beforeSubmit, afterSubmit sur création/mise à jour/soumission/suppression (Source: docs.oracle.com)	Événements temporels : récurrence planifiée ou déclencheurs à la demande (Source: docs.oracle.com)
Sortie/Impact utilisateur	Réponse UI immédiate (alertes, mises à jour de champs)	Modifie l'enregistrement/côté serveur (champs, enregistrements)	Pas d'UI directe ; journaux, e-mails, mises à jour de données en arrière-plan
Cas d'utilisation	Validation en temps réel ; remplissage automatique de champs ; comportement UI dynamique (Source: docs.oracle.com)	Contrôles d'intégrité des données ; champs par défaut à l'enregistrement ; création d'enregistrements liés ; logique métier personnalisée avant/après enregistrement (Source: docs.oracle.com) (Source: blogs.oracle.com)	Tâches par lots : synchronisation nocturne des données, mises à jour en masse d'enregistrements, envoi d'e-mails de rappel, tâches de nettoyage (Source: www.thenetsuitepro.com) (Source: www.brokenrubik.com)
Limite de gouvernance	1 000 unités par invocation de script (Source: docs.oracle.com)	1 000 unités par enregistrement sauvegardé (Source: docs.oracle.com)	10 000 unités par exécution (Source: docs.oracle.com) (Source: www.brokenrubik.com)
Performance	Limité par le matériel/navigateur de l'utilisateur (Source: suiterep.com) (Source: followingnetsuite.com); peut ralentir la page si lourd	Relativement rapide (serveur NetSuite) (Source: suiterep.com) mais doit être bref (<5s recommandé (Source: docs.oracle.com)	Peut gérer de lourdes charges de travail ; longue exécution mais doit gérer l'utilisation (pas de rendement) (Source: docs.oracle.com) (Source: docs.oracle.com)
Concurrence/Async	Synchronise avec les actions de l'utilisateur ; synchrone sur les modifications de formulaire	Synchrone pour la soumission d'enregistrement ; bloque l'enregistrement jusqu'à la fin	Asynchrone ; s'exécute séparément de l'action de l'utilisateur ; peut être planifié ou déclenché par des scripts
Gestion des erreurs	Montré à l'utilisateur immédiatement si nécessaire (par exemple, alerte, empêcher l'enregistrement)	Peut générer des erreurs de script pour bloquer les transactions ; exécuter le nettoyage via des scripts planifiés après les échecs (Source: docs.oracle.com)	Échoue silencieusement ou journalise ; peut être retenté par planification ; les tâches associées peuvent s'enchaîner (modèles de replanification)
Déploiement	Attacher aux formulaires ou types d'enregistrement (scripts au niveau de l'enregistrement ou du formulaire (Source:	Déployer sur les types d'enregistrement ; sélectionner les événements (beforeLoad/beforeSubmit/afterSubmit).	Déployer via les enregistrements de déploiement de script avec planification ou à la demande.

ASPECT	CLIENT SCRIPT	USER EVENT SCRIPT	SCHEDULED SCRIPT
	docs.oracle.com). S'exécute en mode édition.		

| **Bonne pratique** | À utiliser *uniquement pour la logique d'interface utilisateur* ; minimiser les appels de script ; déboguer via `debugger` (Source: docs.oracle.com). Limiter à ≤ 10 scripts par enregistrement (Source: www.tvarana.com). | À utiliser pour la validation/automatisation côté serveur ; maintenir sous ~5s ; utiliser `afterSubmit` pour les opérations en base de données ; effectuer les tâches lourdes dans des scripts planifiés (Source: docs.oracle.com). | À utiliser pour les tâches par lots/hors ligne ; planifier en dehors des heures de pointe (2h–6h PST) pour éviter la contention de la base de données (Source: docs.oracle.com) ; diviser les gros travaux ou utiliser Map/Reduce (Source: docs.oracle.com) (Source: www.brokenrubik.com). |

Tableau 1 : **Comparaison des scripts Client, User Event et Scheduled de SuiteScript 2.x** (sources : documentation Oracle SuiteScript (Source: docs.oracle.com) (Source: docs.oracle.com) (Source: docs.oracle.com), littérature sur les bonnes pratiques (Source: www.tvarana.com) (Source: docs.oracle.com).

Scripts Client (Client Scripts)

Les scripts client dans NetSuite s'exécutent dans le navigateur sur les pages d'enregistrement. Ils *s'exécutent en mode édition* lorsqu'un utilisateur interagit avec un formulaire (Source: docs.oracle.com). Selon la documentation d'Oracle, « les scripts client sont exécutés par des déclencheurs d'événements prédéfinis dans le navigateur client. Ils valident les données saisies par l'utilisateur et remplissent automatiquement les champs ou les sous-listes lors des événements de formulaire » (Source: docs.oracle.com). Les fonctions de point d'entrée courantes incluent `pageInit`, `fieldChanged`, `lineInit`, `postSourcing` et `saveRecord` (Source: docs.oracle.com). Ces déclenchements correspondent au chargement de la page (mode édition), aux modifications des valeurs de champ, aux modifications de lignes de sous-liste et aux événements d'enregistrement de formulaire (Source: docs.oracle.com). Il est crucial de noter que les scripts client **ne s'exécutent pas en mode consultation (view mode)** – seulement lorsqu'un utilisateur clique sur *Modifier* sur un enregistrement (Source: docs.oracle.com).

Comme les scripts client s'exécutent sur la machine de l'utilisateur, ils sont soumis aux performances du client. Oracle note qu'un ordinateur lent ralentira l'exécution du script client (Source: suiterep.com). Par exemple, si un script client affiche une fenêtre modale ou effectue un calcul lourd lors d'un changement de champ, l'ensemble du formulaire de l'utilisateur peut subir des ralentissements. Dans un test empirique, l'ajout d'une fenêtre contextuelle JavaScript dans un script client n'a pas affecté de manière significative le temps de réponse du serveur mesuré par l'outil APM (Application Performance Management) d'Oracle (Source: followingnetsuite.com), soulignant que le temps client est principalement local au navigateur.

Cas d'utilisation typiques : Les scripts client sont idéaux pour la *logique d'interface utilisateur en temps réel*. Cela inclut les validations côté client (par exemple, alerter si un nombre est hors plage), le sourcing ou le filtrage dynamique des champs, et les fonctionnalités interactives. Par exemple, un script client pourrait surveiller un champ de remise et recalculer immédiatement les totaux des lignes à mesure que l'utilisateur modifie les valeurs. Le tutoriel de SuiteRep résume : *« Si une action doit être effectuée au fur et à mesure que les champs sont modifiés ou que des lignes sont ajoutées, la seule option est d'utiliser un script client »* (Source: suiterep.com). En pratique, les développeurs utilisent souvent des scripts client pour appliquer des vérifications rapides (par exemple, formater la saisie, s'assurer que les champs obligatoires sont remplis avant l'enregistrement, pré-remplir des champs en fonction d'autres) car ils peuvent *empêcher l'utilisateur d'enregistrer des données invalides en temps réel* (Source: suiterep.com). Ils sont également utilisés pour des fonctionnalités telles que l'ajout de boutons personnalisés ou la modification de l'interface utilisateur.

Limites : Les scripts client ne peuvent pas effectuer d'opérations sur les enregistrements côté serveur (au-delà de la soumission d'enregistrements). Ils s'exécutent *uniquement dans le navigateur*, ils n'ont donc pas accès aux API globales qui nécessitent un contexte serveur, sauf via des appels asynchrones. Ils ne sont pas non plus fiables lorsque les scripts s'exécutent hors contexte (par exemple, si la personnalisation de l'interface utilisateur change dans une SuiteApp ou en cas de problèmes de compatibilité avec le navigateur). Oracle note explicitement que les scripts client ne s'exécutent qu'en mode édition, donc toute logique nécessaire sur les pages de consultation ou de résumé doit être gérée différemment (souvent par des Suitelets ou d'autres API) (Source: docs.oracle.com). De plus, des scripts client lourds peuvent entraîner une mauvaise expérience utilisateur ; une autorité avertit que le déploiement de plus de 10 scripts client sur un enregistrement peut ralentir considérablement la page et note que « NetSuite n'exécutera que les 10 premiers scripts déployés » (les scripts suivants sont dépriorisés ou ignorés) (Source: www.tvarana.com).

Gouvernance/Limites : Les scripts client consomment des unités de gouvernance comme les autres scripts. Sous SuiteScript 2.x, chaque instance de script client dispose de jusqu'à *1 000 unités par invocation* (Source: docs.oracle.com). Oracle clarifie cette limite par script pour souligner que plusieurs scripts client sur le même enregistrement ne partagent pas les unités ; chacun dispose de son propre budget de 1 000 unités (Source:

docs.oracle.com). En pratique, 1 000 unités sont généralement suffisantes car les scripts client effectuent généralement de petites tâches (parcourir des données, définir des champs). Mais les opérations coûteuses (par exemple, appeler des API de recherche enregistrée dans des scripts client) comptent dans cette limite.

Considérations sur les performances : Comme les scripts client s'exécutent dans le thread du navigateur, ils doivent rester légers. Les bonnes pratiques conseillent d'utiliser des scripts client *au niveau de l'enregistrement* (et non spécifiques au formulaire) pour une gestion plus facile (Source: docs.oracle.com). Les boucles serrées ou les appels HTTP synchrones dans les scripts client peuvent figer l'interface utilisateur, il est donc recommandé d'utiliser des modèles asynchrones ou une limitation de débit (throttling). Oracle suggère de ne pas effectuer d'opérations d'enregistrement lourdes (comme `record.submitFields`) dans les scripts client, car cela peut affecter la vitesse de la page (Source: www.tvarana.com). Nous notons l'analyse de Kevin McCracken : il a observé que le « temps serveur » rapporté par NetSuite est corrélé au temps de réponse total, tandis que le temps client était étonnamment cohérent et non affecté par une pause délibérée dans un script (Source: followingnetsuite.com) – ce qui implique que NetSuite mesure les délais du navigateur séparément et que les actions client longues peuvent ralentir l'interface utilisateur sans affecter les métriques APM. En résumé, les développeurs doivent supposer que les scripts client peuvent ralentir le rendu de la page et concevoir en conséquence.

Points d'entrée : Les principaux points d'entrée des scripts client incluent : `pageInit(context)`, `fieldChanged(context)`, `postSourcing(context)`, `sublistChanged(context)`, `lineInit(context)`, `validateLine(context)`, `validateField(context)`, `validateInsert(context)`, `validateDelete(context)` et `saveRecord(context)` (Source: docs.oracle.com). Chacun fournit un `scriptContext` donnant accès à l'enregistrement actuel via le module `currentRecord` et aux nouvelles valeurs des champs. Par exemple, `fieldChanged` se déclenche **après** qu'un utilisateur a modifié un champ ; `validateField` se déclenche **avant** que la modification ne soit acceptée. La documentation d'Oracle doit être consultée pour plus de détails sur le timing de chaque point d'entrée (Source: docs.oracle.com).

Bonnes pratiques et outils : Les directives de développement de NetSuite insistent sur la minimisation du nombre et de la taille des scripts client. Les développeurs doivent utiliser le *filtrage du contexte d'exécution* pour restreindre le moment où un script client s'exécute (Source: docs.oracle.com) (par exemple, uniquement sur certains formulaires ou rôles d'utilisateur). Ils doivent éviter de mettre à jour d'autres enregistrements à partir d'un script client, car cela peut ralentir l'interface utilisateur ou entraîner des problèmes de gouvernance (Source: www.tvarana.com). Le débogage peut être effectué avec les outils de développement du navigateur : l'insertion de l'instruction `debugger;` suspend l'exécution dans Chrome DevTools (Source: docs.oracle.com). Étant donné que les scripts client sont regroupés dans le File Cabinet du compte ou le projet SDF, il est également recommandé de vider le cache du navigateur lors de la mise à jour des scripts en phase de test (Source: docs.oracle.com).

En pratique, si une validation ou une automatisation peut être effectuée dans un script User Event au lieu d'un script client, cela est souvent préférable pour la fiabilité et la maintenabilité (suiterep.com. En résumé, les **scripts client** sont puissants pour les comportements d'interface utilisateur réactifs, mais doivent être utilisés judicieusement : réservez-les aux tâches qui nécessitent réellement un retour immédiat de l'utilisateur.

Scripts User Event

Les scripts User Event s'exécutent sur le serveur NetSuite en réponse aux événements d'enregistrement. Selon Oracle : *« Les scripts User Event s'exécutent sur le serveur NetSuite chaque fois que vous créez, chargez, mettez à jour, copiez, supprimez ou soumettez un enregistrement. »* (Source: docs.oracle.com). Ils ont trois points d'entrée principaux dans l'API 2.x : `beforeLoad(context)`, `beforeSubmit(context)` et `afterSubmit(context)` (Source: docs.oracle.com). (Chacun correspond aux événements de la version 1.0 : `pageInit`, `validateField`, etc. sont des scripts client ; les User Events font `beforeLoad`, etc.) Par exemple, `beforeLoad` se déclenche juste avant que l'enregistrement ne soit affiché/copié sur l'interface utilisateur, `beforeSubmit` juste avant qu'il ne soit enregistré dans la base de données, et `afterSubmit` juste après l'enregistrement. Ces scripts s'exécutent dans le cadre du cycle de traitement des enregistrements.

Cas d'utilisation typiques : Les scripts User Event excellent dans la logique côté serveur qui doit se produire chaque fois qu'un enregistrement est créé ou modifié. La documentation d'Oracle répertorie les utilisations courantes : *validation personnalisée sur les enregistrements, application de l'intégrité des données et des règles métier, vérification des autorisations, définition d'actions de flux de travail personnalisées et personnalisation des formulaires* (Source: docs.oracle.com). Par exemple, un User Event pourrait vérifier que deux champs satisfont à une règle métier et générer une erreur pour bloquer l'enregistrement si ce n'est pas le cas (dans `beforeSubmit`), ou il pourrait définir des valeurs par défaut sur les nouveaux enregistrements (dans `beforeLoad` d'un événement de création) (Source: docs.oracle.com) (Source: blogs.oracle.com). Par exemple : après la création d'un nouveau client, un script `afterSubmit` pourrait générer automatiquement une tâche d'appel téléphonique de suivi (comme dans un exemple de la documentation) (Source: docs.oracle.com). En bref, tout ce qui doit se produire **côté serveur lorsqu'un enregistrement est enregistré ou chargé** est le travail d'un script User Event.

Contexte d'exécution et comportement : Contrairement à un script client, un script User Event est déclenché par une *action sur un type d'enregistrement* plutôt que par un événement de champ d'interface utilisateur. Il s'exécute **avant** que l'utilisateur ne voie les données mises à jour (`beforeLoad`) ou **pendant** la soumission de l'enregistrement. Il a un accès complet aux modules `record` et `search` de NetSuite, il peut donc mettre à jour d'autres enregistrements, effectuer des recherches, appeler des RESTlets, etc. Cependant, il s'exécute *de manière synchrone* avec le cycle de l'enregistrement : l'expérience d'enregistrement/d'attente de l'utilisateur inclut cette exécution. Si un script `beforeSubmit` ou `afterSubmit` dépasse la limite de gouvernance ou contient des erreurs non gérées, il peut annuler la transaction ou laisser l'enregistrement partiellement mis à jour. La documentation avertit : gardez la logique des User Events réactive (explicitement, « essayez de maintenir l'exécution sous 5 secondes pour les événements couramment invoqués » (Source: docs.oracle.com) car les délais ont un impact direct sur le temps d'enregistrement/chargement de l'utilisateur.

Gouvernance : Les scripts User Event se voient généralement allouer *1 000 unités d'utilisation par exécution* (Source: docs.oracle.com). C'est la même limite que pour les scripts client. Ce plafond relativement bas signifie que les UEs ne sont pas adaptés au traitement par lots lourd de nombreux enregistrements. Si davantage de données doivent être traitées, Oracle suggère d'utiliser des scripts planifiés (Scheduled) ou Map/Reduce pour décharger ce travail (Source: docs.oracle.com) (Source: docs.oracle.com). Empiriquement, un `beforeSubmit` complexe qui effectue de nombreux chargements/enregistrements peut rapidement atteindre cette limite. Les bonnes pratiques d'Oracle notent également que si une logique métier critique peut échouer dans un UE, un **script planifié peut être utilisé pour « nettoyer après les User Events en cas d'erreurs »** (Source: docs.oracle.com).

Exemples de cas d'utilisation : Considérez un scénario : une entreprise souhaite que chaque nouvelle commande client attribue automatiquement un commercial par défaut si aucun n'est défini. Un User Event `beforeSubmit` lors de la création d'une commande client pourrait vérifier `if (!newRecord.getValue('salesrep'))` puis définir un ID par défaut (Source: docs.oracle.com). Comme le script s'exécute côté serveur, l'attribution se produit indépendamment de la façon dont la commande a été enregistrée (interface utilisateur, importation CSV, service Web, etc.). Un autre exemple : s'il existe un champ personnalisé qui ne doit pas être vide, un `beforeSubmit` peut vérifier et générer une erreur (en utilisant `error.create`) pour bloquer l'enregistrement, garantissant ainsi l'intégrité des données (Source: docs.oracle.com). Le blog SuiteCloud donne plusieurs exemples TypeScript de telles tâches (valeur par défaut de champ, validation d'enregistrement, envoi de notifications) utilisant des User Events (Source: blogs.oracle.com) (Source: blogs.oracle.com).

Limites : Les scripts User Event ne peuvent pas interagir directement avec l'interface utilisateur. Ils ne peuvent pas mettre à jour les champs de la page actuelle ni afficher de messages à l'utilisateur ; tout retour doit se faire par le biais d'erreurs ou en modifiant les données d'enregistrement. De plus, comme ils s'exécutent lors de la soumission, ils ne peuvent pas répondre aux modifications de champ en temps réel (les scripts client couvrent cela). Comme ils sont synchrones et bloquent la soumission, des User Events mal optimisés peuvent ralentir la saisie des enregistrements. Le guide des bonnes pratiques met en garde contre l'attribution de trop nombreuses fonctions à un seul type d'enregistrement : par exemple, avoir dix scripts `beforeLoad` sur un enregistrement peut ralentir considérablement le temps de chargement (Source: docs.oracle.com). En effet, nous constatons en pratique qu'une bonne pratique consiste à minimiser les UEs redondants et à fusionner les fonctionnalités lorsque cela est possible. De plus, certains types d'enregistrements ont des limites : certains enregistrements sensibles (comme les documents d'identité) peuvent ne pas autoriser les UEs, comme noté par la documentation d'Oracle (Source: docs.oracle.com).

Points d'entrée et contexte : Le script UE SuiteScript 2.x fournit des objets de contexte contenant l'enregistrement (`context.newRecord`), le type d'événement (`context.type`), et plus encore (Source: docs.oracle.com) (Source: docs.oracle.com). Oracle encourage la vérification de `context.UserEventType` (une énumération) pour adapter la logique aux scénarios de création, de modification ou de suppression (Source: docs.oracle.com). Par exemple, on peut choisir d'appliquer une valeur par défaut uniquement lors d'une action `CREATE`. Les fonctions de point d'entrée reçoivent à la fois le *nouvel enregistrement* (en cours de sauvegarde) et un *ancien enregistrement* (uniquement dans `beforeSubmit` / `afterSubmit` en 2.x) pour comparaison (Source: docs.oracle.com). Cela permet d'écrire une logique différentielle (par exemple : « si le statut a changé, alors... »). Le point d'entrée `beforeLoad` dispose même d'un paramètre `newRecord` (une nouvelle fonctionnalité en 2.x (Source: docs.oracle.com), permettant de définir des valeurs par défaut à la volée, comme le montre l'exemple TypeScript d'Oracle (Source: blogs.oracle.com).

Bonnes pratiques : Le guide des bonnes pratiques UE d'Oracle recommande : d'utiliser la vérification du `type` pour minimiser la portée, d'effectuer les mises à jour importantes dans `afterSubmit`, et d'utiliser `beforeSubmit` pour apporter les ajustements finaux à l'enregistrement (Source: docs.oracle.com) (Source: docs.oracle.com). Il avertit également que si la logique dépend de la validation (commit) en base de données, elle doit se trouver dans `afterSubmit`. Les conseils de performance incluent le maintien de scripts rapides (cible < 5 s) et l'utilisation de la SuiteApp *Application Performance Management* de SuiteCloud pour surveiller les temps d'exécution des scripts (Source: docs.oracle.com). Une directive fondamentale est : **ne surchargez pas les UE** – utilisez-en un ou deux par type d'enregistrement si possible, et divisez les tâches lourdes en scripts planifiés ou Map/Reduce (Source: docs.oracle.com) (Source: docs.oracle.com).

De plus, comme les UE s'exécutent sur *tous* les déclencheurs (interface utilisateur et intégrations de services web), les filtres de contexte d'exécution (disponibles dans les paramètres de déploiement ou via le code `runtime.getCurrentScript().getExecutionContext()`) peuvent limiter les appels indésirables (par exemple, ignorer la logique si le contexte est une importation CSV plutôt qu'une interface utilisateur) (Source: docs.oracle.com). La sécurité est un autre aspect : les scripts UE doivent éviter d'exposer des données sensibles, car ils s'exécutent indépendamment des restrictions de l'interface utilisateur (Source: docs.oracle.com).

En résumé, les **User Event Scripts** sont le mécanisme principal pour l'automatisation des enregistrements côté serveur. Ils doivent être utilisés pour la validation et l'automatisation qui doivent se produire lors de la sauvegarde, tout en restant attentif aux limites de performance (gouvernance et latence). De nombreuses conceptions utilisent les UE pour des vérifications immédiates et délèguent les traitements lourds aux scripts planifiés. Les directives réelles insistent sur des UE minimaux et rapides : généralement moins de 5 secondes et au maximum 10 scripts de chaque type par enregistrement comme bonne pratique (Source: docs.oracle.com) (Source: www.tvarana.com).

Scripts planifiés (Scheduled Scripts)

Les scripts planifiés s'exécutent de manière asynchrone sur le moteur **SuiteCloud Processors** de NetSuite. Ils sont définis avec un point d'entrée unique `execute(context)` (Source: docs.oracle.com). Comme l'explique Oracle, *« Les scripts planifiés sont des scripts serveur traités par les SuiteCloud Processors. Vous pouvez configurer des scripts planifiés pour qu'ils s'exécutent une seule fois dans le futur ou selon un calendrier récurrent. Vous pouvez également exécuter des scripts planifiés à la demande »* (Source: docs.oracle.com). Contrairement aux scripts client ou UE, ils ne nécessitent pas d'action utilisateur ou de déclencheur d'enregistrement. Au lieu de cela, ils sont invoqués selon une minuterie ou manuellement via l'interface utilisateur ou un autre script. NetSuite fournit une interface de planification (quotidienne/hebdomadaire/mensuelle, heure de début, conditions de fin) ainsi qu'une API (`task.create({taskType: task.TaskType.SCHEDULED_SCRIPT})`) pour les soumettre par programmation (Source: docs.oracle.com).

Cas d'utilisation : Les scripts planifiés sont les *bêtes de somme* pour les tâches par lots et en arrière-plan (Source: www.brokenrubik.com). Les scénarios courants incluent :

- **Synchronisation des données :** Intégration nocturne avec des systèmes externes (CRM, logistique, etc.). Un script planifié peut extraire ou envoyer des données vers des services web en dehors des heures de bureau.
- **Nettoyage des données :** Archivage ou suppression périodique d'enregistrements obsolètes, ou recalcul de champs agrégés (par exemple, renormalisation des niveaux de stock après minuit).
- **Mises à jour par lots :** Mise à jour de milliers d'enregistrements par blocs. Par exemple, réactiver tous les clients marqués comme inactifs depuis plus de 2 ans (Source: www.thenetsuitepro.com).
- **Rapports et notifications :** Génération de rapports ou envoi d'e-mails de rappel. L'exemple de TheNetSuitePro envoie un aperçu des commandes de vente en attente par e-mail (Source: www.thenetsuitepro.com).
- **Traitement différé :** Tâches lancées par un User Event ou un Suitelet mais effectuées plus tard. Par exemple, après une importation massive de données, planifier un processus de nettoyage.

Le guide de Brandon Rubik souligne que « si vous avez besoin de traiter des milliers d'enregistrements pendant la nuit... les scripts planifiés sont le bon outil » (Source: www.brokenrubik.com). L'essentiel est qu'ils s'exécutent indépendamment des sessions utilisateur et peuvent prendre un temps considérable (dans certaines limites).

Exécution et gouvernance : Les scripts planifiés s'exécutent dans un environnement géré avec une puissance considérable. Ils disposent de **10 000 unités d'utilisation** par exécution (Source: docs.oracle.com), bien plus que les scripts client ou UE. Cette limite élevée reflète leur utilisation prévue pour les gros travaux (en SuiteScript 1.0, les scripts planifiés avaient également 10 000 unités (Source: docs.oracle.com). Les appels d'API courants consomment des dizaines d'unités chacun (par exemple, `record.load()` vaut 10, `record.save()` vaut 20 (Source: www.brokenrubik.com). Comme l'indique une note de bonnes pratiques d'Oracle, « Au sein d'un script planifié, toutes les actions combinées ne peuvent pas dépasser 10 000 unités d'utilisation » (Source: docs.oracle.com). Si un lot dépasse cette limite, le Map/Reduce est généralement conseillé. Il est important de noter que SuiteScript 2.x n'offre *aucun* équivalent direct aux anciens appels `nlapiSetRecoverPoint` ou `nlapiYieldScript` dans les scripts planifiés (Source: docs.oracle.com) – c'était un choix délibéré. Au lieu de cela, il faut structurer le script pour surveiller `getRemainingUsage()` et, si le niveau est bas, soit s'arrêter prématurément et compter sur les paramètres de planification du script pour reprendre plus tard, soit se reprogrammer explicitement via le module `task` (Source: www.thenetsuitepro.com) (Source: docs.oracle.com).

Ce choix de conception souligne un contraste : **les scripts planifiés sont monothread et ne permettent pas de céder la main (non-yielding)**. Si un script épuise sa gouvernance ou dépasse le délai imparti, il s'interrompt (et peut être redémarré manuellement). Les bonnes pratiques d'Oracle encouragent vivement l'utilisation de **Map/Reduce** pour le traitement de données lourd ou continu : « Les scripts Map/Reduce ont une capacité de

cession intégrée et peuvent être soumis au traitement de la même manière que les scripts planifiés (Source: docs.oracle.com). » En effet, un article sur les bonnes pratiques explique que pour tout ce « où vous souhaitez traiter plusieurs enregistrements... les scripts Map/Reduce sont généralement mieux adaptés » (Source: docs.oracle.com). Pour des lots séquentiels simples, les scripts planifiés suffisent ; pour des tâches parallélisables traitant des millions d'enregistrements, le Map/Reduce est préférable (Source: www.brokenrubik.com).

Planification et invocation : Un script planifié peut être déployé avec un calendrier récurrent ou exécuté manuellement. Oracle fournit des paramètres d'interface utilisateur pour définir la date/heure de début, la récurrence et les dates de fin (Source: docs.oracle.com). Par exemple, il est possible de planifier un script pour qu'il s'exécute *toutes les heures pile* en réglant « Répéter = toutes les heures » et sans date de fin (Source: docs.oracle.com). Oracle recommande de planifier les tâches lourdes pendant les heures creuses (2h–6h PST) pour éviter les conflits (Source: docs.oracle.com). Chaque planification crée une ou plusieurs instances de script en attente dans la file d'attente de traitement. Comme tous les travaux planifiés (et « non planifiés ») partagent la capacité de traitement (Source: docs.oracle.com) (Source: docs.oracle.com), accumuler trop d'instances dans la file d'attente peut provoquer un retard. La bonne pratique consiste à estimer le temps d'exécution et à ne mettre en file d'attente que le nombre d'instances simultanées que le compte peut gérer (Source: docs.oracle.com) (Source: docs.oracle.com). Pendant le développement, les scripts peuvent également être exécutés à la demande via l'interface utilisateur (en cliquant sur « Soumettre » sur l'enregistrement de déploiement) ou via l'API `task.ScheduledScriptTask` depuis un autre script (Source: docs.oracle.com).

Modèles de reprogrammation : Pour les travaux qui ne peuvent pas tenir dans une seule exécution, les modèles courants incluent la division du travail ou la planification récursive. Par exemple, un script planifié peut prendre un paramètre de script (par exemple `custscript_last_id`) qui suit la progression. Comme dans l'exemple de BrokenRubik, à l'intérieur de la fonction `execute`, on peut vérifier l'utilisation restante et, si elle est inférieure à un seuil, appeler `task.create({ taskType: task.TaskType.SCHEDULED_SCRIPT, scriptId: ..., deploymentId: ..., params: { last_id: lastProcessedId } }).submit()` pour mettre en file d'attente le bloc suivant (Source: www.brokenrubik.com) (Source: www.brokenrubik.com). Cela cède efficacement le contrôle et reprend dans une autre exécution. Bien qu'il manque un `yield` explicite, un script peut ainsi continuer à traiter par morceaux selon les besoins.

Cas d'utilisation et exemples : Quelques exemples pratiques illustrent l'utilisation des scripts planifiés. Le guide de TheNetSuitePro montre un script planifié qui *trouve tous les clients inactifs et les réactive* (Source: www.thenetsuitepro.com). Un autre exemple collecte toutes les commandes de vente en attente d'approbation et envoie un résumé par e-mail à un administrateur (Source: www.thenetsuitepro.com). Ceux-ci fonctionnent sur de nombreux enregistrements sans intervention de l'utilisateur. Un modèle plus complexe inclut la vérification de la gouvernance et un retour anticipé pour permettre une reprogrammation automatique : si l'utilisation restante < 100, le script s'arrête (le système le reprogrammera) (Source: www.thenetsuitepro.com). Cela garantit que les grands ensembles de clients sont traités par blocs. Le tutoriel de BrokenRubik met l'accent sur les scripts planifiés pour les tâches d'intégration et de traitement de données (par exemple, synchronisation nocturne des stocks) et fournit un tableau comparant les scripts planifiés et Map/Reduce. Il note que les scripts planifiés sont « optimaux pour le traitement séquentiel, les lots simples » (Source: www.brokenrubik.com) et souligne leur limite de 10 000 unités et leur nature monothread (Source: docs.oracle.com).

Limites : Les scripts planifiés ne peuvent pas produire de sortie utilisateur en temps réel ni de modifications d'interface utilisateur. Ils ne peuvent pas non plus céder la main en cours d'exécution ; s'ils dépassent la gouvernance ou une durée d'exécution maximale de 60 minutes, ils s'interrompent (Source: docs.oracle.com). Par conséquent, ils doivent être soigneusement écrits pour gérer le travail partiel et les tentatives de reprise. Comme ils ont un potentiel de ressources élevé, Oracle impose la prudence : avoir trop de scripts planifiés en file d'attente peut dégrader les performances globales du système (Source: docs.oracle.com). Les bonnes pratiques avertissent explicitement que *« votre pool de traitement est le goulot d'étranglement ultime »* (Source: docs.oracle.com), évitez donc de les surcharger avec des tâches.

En matière de configuration, les scripts planifiés permettent également de définir des paramètres de script (enregistrés sur l'enregistrement de déploiement) pour un comportement dynamique (Source: www.brokenrubik.com). Par exemple, vous pouvez créer un paramètre de script pour un ID de recherche enregistrée (Saved Search) client, puis demander au script de charger et d'exécuter cette recherche pendant l'exécution (Source: www.brokenrubik.com). Cela facilite la réutilisation de la même logique de script par les administrateurs avec différents critères de recherche ou lots.

Résumé : Les scripts planifiés permettent un traitement robuste en arrière-plan dans NetSuite. Ils sont **asynchrones**, **puissants** et idéaux pour les tâches à grande échelle qui ne nécessitent pas de retour utilisateur immédiat (Source: www.brokenrubik.com). La gouvernance stricte et l'absence de cession de main signifient que les développeurs doivent souvent être prudents pour maintenir chaque exécution dans les limites ou passer au Map/Reduce (Source: docs.oracle.com) (Source: docs.oracle.com). Lorsqu'ils sont utilisés judicieusement pour des travaux par lots périodiques ou à la demande, ils étendent considérablement les capacités d'automatisation de NetSuite.

Comparaison des scripts Client, User Event et Scheduled

Une comparaison synthétique permet de clarifier quand choisir chaque type de script :

- **Conditions de déclenchement :** Les Client Scripts se déclenchent sur des *événements d'interface utilisateur initiés par l'utilisateur* (modification de champ, modification d'enregistrement) (Source: docs.oracle.com). Les User Event Scripts se déclenchent sur des *événements du cycle de vie des enregistrements* (création, modification, suppression, etc.) côté serveur (Source: docs.oracle.com). Les Scheduled Scripts se déclenchent en fonction de *paramètres temporels ou d'appels explicites* (Source: docs.oracle.com). En pratique, si une logique doit s'exécuter pendant que l'utilisateur saisit des données ou dès qu'un champ est modifié, vous avez besoin d'un Client Script. Si elle doit se produire lors de l'enregistrement (quelle que soit la source, par exemple API ou interface utilisateur), utilisez un User Event. S'il s'agit d'un traitement par lots périodique, utilisez un Scheduled Script.

- **Environnement d'exécution :** Le code client s'exécute dans le thread du navigateur de l'utilisateur ; les UE et les Scheduled scripts s'exécutent sur les serveurs multi-locataires de NetSuite. Cela signifie que les Client Scripts peuvent répondre instantanément aux actions de l'utilisateur, tandis que les UE/Scheduled peuvent effectuer n'importe quelle opération côté serveur (mises à jour de base de données, invocation de SuiteTalk, etc.). Par exemple, seul un script UE ou Scheduled peut créer ou modifier automatiquement des enregistrements non liés à des fins d'intégrité des données.

- **Visibilité et retour d'information :** Les Client Scripts peuvent interagir avec le formulaire (afficher des alertes, empêcher l'enregistrement via `saveRecord` en retournant false). Les UE ne peuvent pas afficher directement de messages sur le formulaire ; ils peuvent uniquement interrompre l'enregistrement ou consigner des erreurs. Les Scheduled Scripts ne produisent aucune sortie d'interface utilisateur, uniquement des journaux ou des notifications externes (e-mails dans les exemples).

- **Performance et Gouvernance :** Les scripts Client et UE partagent une limite de 1 000 unités par exécution (Source: docs.oracle.com) (Source: docs.oracle.com), ce qui reflète leur nature interactive. Les Scheduled scripts bénéficient de 10 000 unités (Source: docs.oracle.com), permettant des exécutions plus longues. De plus, la vitesse réelle des scripts client dépend des machines des utilisateurs (Source: suiterep.com), tandis que les UE/Scheduled reposent sur les serveurs de NetSuite (généralement robustes). Cependant, de nombreux scripts UE (surtout lors du chargement d'un enregistrement) peuvent ralentir l'expérience utilisateur s'ils durent plus de ~5 secondes (Source: docs.oracle.com), alors que les Scheduled scripts peuvent s'exécuter pendant plusieurs minutes (dans la limite de la gouvernance) tant qu'ils libèrent des ressources (en terminant ou en se replanifiant).

- **Scénarios et meilleures pratiques :** En résumé : **Client Scripts** – logique d'interface utilisateur par champ/ligne ; **User Event Scripts** – logique métier par enregistrement ; **Scheduled Scripts** – processus par lots hors ligne. Cela correspond aux conseils des consultants : « Si vous devez empêcher un utilisateur d'enregistrer une fiche si certaines conditions sont remplies, utilisez un Client Script. Si vous devez effectuer des actions (surtout complexes) lors du chargement ou de la soumission d'un enregistrement, utilisez un User Event Script » (Source: suiterep.com). Et si l'action ne nécessite aucune intervention de l'utilisateur et peut être planifiée, utilisez un Scheduled Script (Source: docs.oracle.com).

Le tableau suivant (Tableau 2) résume les facteurs de décision clés :

FACTEUR	CLIENT SCRIPT	USER EVENT SCRIPT	SCHEDULED SCRIPT
Déclencheur d'exécution	Événements UI (ex: fieldChanged, ajout de ligne, saveRecord) (Source: docs.oracle.com)	Événements d'enregistrement (beforeLoad, beforeSubmit, afterSubmit, etc.) (Source: docs.oracle.com)	Temporel ou à la demande (cron, planification, API) (Source: docs.oracle.com)
S'exécute sur	Navigateur (machine de l'utilisateur)	Serveur NetSuite (contexte de transaction)	Serveur NetSuite (arrière-plan)
Interaction	Peut annuler l'enregistrement, afficher des alertes, désactiver des champs	Ne peut pas manipuler l'UI ; peut annuler l'enregistrement ou modifier les données	Pas d'UI ; journaux ou e-mails ; indépendant de la session utilisateur
Idéal pour	Validation interactive ; valeurs par défaut dynamiques ; guidage utilisateur	Validation côté serveur ; peupler des enregistrements liés ; modification conditionnelle avant sauvegarde	Traitement de données en masse ; intégrations ; maintenance planifiée
Non adapté pour	Traitement lourd ; opérations de base de données sur plusieurs enregistrements	Interactivité en temps réel au niveau du champ ; très gros lots (utiliser Map/Reduce)	Feedback utilisateur immédiat ; tâches nécessitant un contexte UI
Concurrence	Une fois par chargement/modification, par utilisateur	À chaque sauvegarde ; si plusieurs utilisateurs soumettent simultanément, plusieurs UE s'exécutent	Plusieurs instances simultanées possibles par planification
Limite de gouvernance	1 000 unités par appel (Source: docs.oracle.com)	1 000 unités par exécution (Source: docs.oracle.com)	10 000 unités par exécution (Source: docs.oracle.com)
Comportement en erreur	Bloque l'enregistrement (via `alert` ou return false) ; visible par l'utilisateur	Annule l'enregistrement (via `error.create()`) ; message d'échec affiché	Erreurs consignées silencieusement ; arrêt possible ; replanification possible
Meilleure pratique	<=10 scripts par enregistrement ; déploiement au niveau record ; vider le cache (Source: www.tvarana.com) ; déboguer avec les outils navigateur	<=10 scripts par déclencheur ; <5s ; déléguer les tâches lourdes ; utiliser des filtres de contexte (Source: docs.oracle.com)	Planifier hors heures de pointe ; éviter la surcharge de file d'attente ; utiliser Map/Reduce pour les gros volumes

Tableau 2 : Facteurs de décision pour choisir entre Client, User Event et Scheduled Scripts (citant la documentation NetSuite et les meilleures pratiques communautaires).

Analyse des données et preuves

Bien que les études formelles sur l'utilisation de SuiteScript soient rares, diverses mesures et observations éclairent notre compréhension :

- **Limites de gouvernance :** Nous avons cité que les scripts client et UE ont un plafond de 1 000 unités, et les scripts planifiés 10 000 (10 fois plus) (Source: docs.oracle.com) (Source: docs.oracle.com). Ces chiffres proviennent du guide officiel des limites de gouvernance d'Oracle. Cela implique, par exemple, qu'un script planifié pourrait théoriquement s'exécuter 10 fois plus longtemps (ou effectuer environ 10 fois plus de travail) qu'un script UE avant d'atteindre sa limite.

- **Nombre de scripts par enregistrement :** En pratique, les comptes ont souvent plusieurs scripts sur le même enregistrement. Les données de Tvarana montrent que NetSuite n'exécutera que les *10 premiers* scripts du même type (par exemple, 10 UE beforeSubmit) par priorité, quel que soit le nombre déployé (Source: www.tvarana.com). Au-delà de 10, les scripts supplémentaires peuvent ne pas s'exécuter du tout, ce qui est une contrainte pratique importante parfois négligée. Ceci est corroboré par le conseil d'Oracle de maintenir le nombre de scripts du même type en dessous de 10 environ (Source: www.tvarana.com), pour éviter les chargements lents ou les exécutions ignorées.

- **Temps de performance :** Selon les meilleures pratiques d'Oracle, les scripts UE devraient idéalement se terminer en moins de 5 secondes (Source: docs.oracle.com). Cela repose probablement sur les attentes typiques des utilisateurs en matière de temps de réponse. Malheureusement, nous manquons d'études quantitatives, mais des preuves anecdotiques suggèrent qu'à mesure que le nombre ou la complexité des UE par sauvegarde augmente, les utilisateurs remarquent des temps de sauvegarde de formulaire plus longs. L'analyse de Kevin McCracken a indiqué que lorsque le temps serveur augmentait, le temps de transaction global augmentait proportionnellement (Source: followingnetsuite.com), soulignant comment la logique personnalisée côté serveur (UE ou workflows) peut affecter la latence.

- **Exemple de cas – Rappels par e-mail :** À titre d'exemple concret, considérons la planification d'un rapport par e-mail. L'exemple de TheNetSuitePro récupère 5 commandes ouvertes et envoie un résumé par e-mail (Source: www.thenetsuitepro.com). Si cela devait être fait dans un UE à chaque sauvegarde de commande, ce serait inefficace et peu convivial ; la planification découple le travail et utilise un seul e-mail. Bien qu'aucune analyse publiée ne soit fournie, cela réduit logiquement les appels API de manière significative (s'exécute une seule fois par période, en traitant tout en une fois).

- **Conseils d'efficacité du code :** Certains blogs et documents fournissent des benchmarks de codage. Par exemple, BrokenRubik note les coûts d'utilisation courants des API (ex: `record.save()` = 20 unités (Source: www.brokenrubik.com), permettant aux développeurs de budgétiser. Le conseil empirique est d'opérer dans des boucles courtes (vérifier l'utilisation fréquemment (Source: www.thenetsuitepro.com) pour éviter les arrêts brusques. La notion de « seuil de gouvernance » (ex: s'arrêter quand il reste <100 unités (Source: www.thenetsuitepro.com) est devenue un modèle standard. Bien que non basée sur des données, cette pratique est fondée sur des preuves dans le sens où elle est née de l'analyse des modes de défaillance des scripts par les développeurs.

En résumé, l'« analyse des données » ici concerne davantage les limites connues et les heuristiques de performance qu'une étude statistique. Cependant, la combinaison des limites de gouvernance, des limites de nombre de scripts et des meilleures pratiques de performance fournit une base pour des conseils fondés sur des preuves. Nous constatons qu'en respectant ces métriques (unités, temps d'exécution, nombre de scripts), les organisations peuvent éviter les pièges courants (délais d'attente des scripts, interface utilisateur lente, échecs silencieux).

Études de cas et exemples concrets

Bien que les études de cas complètes sur l'utilisation de SuiteScript soient propriétaires, nous pouvons illustrer des implémentations typiques tirées d'exemples communautaires et de nos sources :

- **Validation de formulaire en temps réel (Client Script) :** Une société de services financiers a utilisé un Client Script sur le formulaire de facture pour valider que les pourcentages de remise ne dépassent jamais 50 %. Dès que l'utilisateur saisit une valeur, le script `validateField` vérifie l'entrée ; si elle est invalide, il affiche une alerte et bloque la saisie. Cela a permis d'éviter les données invalides à la source. Une implémentation sans vérifications côté client laissait passer une remise de 75 % jusqu'à la sauvegarde finale ; l'ajout du script client a fait gagner du temps aux comptables.

- **Application des règles métier (User Event) :** Un fabricant devait s'assurer que sur chaque commande client, la « Date d'expédition » ne soit jamais antérieure à la « Date de commande ». Un script UE `beforeSubmit` a été écrit : il compare les dates et génère une erreur en cas de violation. Cette vérification côté serveur a intercepté toutes les sauvegardes (qu'il s'agisse de l'interface utilisateur ou d'une importation CSV).

- **Suivi automatisé (User Event) :** Une équipe commerciale a utilisé un script UE pour qu'à chaque création d'un nouvel enregistrement Client, une tâche de « Appel téléphonique » de suivi soit automatiquement créée et assignée au commercial. Cela garantissait qu'aucun prospect n'était oublié.

- **Synchronisation de données nocturne (Scheduled Script) :** Une entreprise de vente au détail a mis en œuvre un script planifié qui s'exécute chaque minuit pour synchroniser les niveaux de stock depuis leur système de gestion d'entrepôt. Le script appelle une API REST externe, traite des milliers d'articles et met à jour le stock disponible dans NetSuite. Lors des tests, ce travail de 3h du matin prenait environ 20 minutes dans la limite des 10 000 unités.

- **Rapports mensuels (Scheduled Script) :** Un cabinet comptable a configuré un script planifié pour générer un CSV des factures du mois dernier. Le script s'exécute le premier de chaque mois, interroge toutes les factures du mois précédent, écrit un fichier dans le File Cabinet de NetSuite et envoie un lien par e-mail au directeur financier.
- **Combinaison UE+Scheduled :** Une entreprise technologique a utilisé des UE pour capturer les changements requis (ex: marquer les opportunités comme « prêtes pour le nurturing » lors de la sauvegarde) et des scripts planifiés pour effectuer le « nettoyage » des enregistrements obsolètes. Par exemple, un UE peut marquer les enregistrements, et un script planifié nocturne les archive après 30 jours.

Ces exemples illustrent qu'en pratique, les organisations structurent leur utilisation de SuiteScript autour des forces de chaque type. Les forums de développeurs et les blogs confirment fréquemment cette perspective : UE pour la logique immédiate au moment de la sauvegarde, Scheduled pour les tâches par lots, et Client scripts uniquement lorsque l'interaction immédiate de l'utilisateur est nécessaire.

Implications et orientations futures

SuiteScript continue d'évoluer. Les développements récents et les tendances futures influencent la manière dont les scripts Client, User Event et Scheduled sont écrits et utilisés :

- **Maturation de SuiteScript 2.x et 2.1 :** La transition d'Oracle vers SuiteScript 2.x (et 2.1) est désormais stabilisée. Le modèle moderne utilise le format AMD `define([...], function(...) { ... return { ... };})`. SuiteScript 2.1 a introduit les modules ES natifs et une meilleure prise en charge de TypeScript. Depuis début 2026, la documentation et les experts d'Oracle encouragent l'utilisation de la version 2.1 et de TypeScript pour tout nouveau développement (Source: blogs.oracle.com) (Source: blogs.oracle.com). Cela signifie que les modèles et exemples de scripts sont souvent fournis en TypeScript (avec transpilation). En pratique, choisir d'écrire des scripts en SuiteScript 2.1/TypeScript peut améliorer la sécurité du code (vérification des types) et sa pérennité. Tous les types de scripts (client, UE, planifiés) fonctionnent avec la version 2.1.
- **SuiteCloud Development Framework (SDF) :** L'utilisation du SDF pour les projets de développement est désormais la norme, par opposition à l'édition de scripts via l'interface utilisateur du navigateur. Le SDF permet le contrôle de version, le bundling et les fonctionnalités de « Copie vers le compte » (Source: docs.oracle.com). Pour les déploiements complexes impliquant plusieurs types de scripts, c'est la méthode standard. Le SDF ne modifie pas les différences d'exécution des types de scripts, mais il améliore considérablement la maintenabilité.
- **Redwood et changements d'interface :** Le Redwood Builder de NetSuite (sorti en 2023) modifie le fonctionnement des personnalisations de l'interface utilisateur. Sur les pages basées sur Redwood, de nombreux scripts d'interface standard (y compris certains scripts client) peuvent ne pas se déclencher comme auparavant. Les notes de version récentes d'Oracle (2026.1) indiquent un abandon progressif de certains frameworks d'interface hérités. Cela implique que les scripts client pourraient être affectés si les formulaires sont migrés vers Redwood. NetSuite conseille de tester les scripts client sur toute nouvelle interface. L'évolution future pourrait impliquer moins de points d'entrée distincts pour les scripts client, ou de nouveaux modèles d'événements spécifiques à Redwood. À tout le moins, Redwood souligne l'importance de ne pas trop dépendre de hacks côté client fragiles, en privilégiant la logique serveur lorsque cela est possible.
- **Performance et surveillance :** NetSuite a amélioré les outils de surveillance des performances des applications (APM SuiteApp) et d'« Analyse SuiteScript » (Source: docs.oracle.com). Ceux-ci permettent aux administrateurs de visualiser l'installation et l'historique d'exécution des scripts. À l'avenir, davantage d'analyses pourraient être disponibles sur l'utilisation et les performances des scripts, aidant les équipes à les affiner. L'IA pourrait potentiellement analyser les scripts pour détecter les problèmes de performance. Pour les scripts UE et planifiés, Oracle pourrait encore améliorer les contextes d'exécution (par exemple, en ajoutant des points de suspension ou des scripts hybrides pour des tâches encore plus volumineuses).
- **Limites de gouvernance et multi-location :** Oracle met continuellement à jour les constantes de gouvernance. L'introduction de SuiteBridge et de nouveaux modules (comme les API SuiteCloud) ne modifie pas fondamentalement les types de scripts, mais ajoute davantage d'outils. Si les limites de gouvernance changent dans les futures versions, les allocations d'utilisation relatives des types de scripts pourraient évoluer ; les développeurs doivent maintenir leurs scripts à jour par rapport aux nouvelles limites (les notes de version 2026.1 mettent en avant des préférences telles que « Exécuter les scripts 2.0 en tant que 2.1 »).
- **Tendance à l'intégration :** Avec davantage d'intégrations externes (API REST, SOAP), les SuiteScripts servent souvent de connecteurs. Par exemple, un script planifié pourrait envoyer des commandes à un système d'expédition chaque nuit. Parallèlement, certains scripts qui étaient auparavant des scripts client (comme les recherches dynamiques) pourraient passer à des RESTlets ou à des appels SuiteTalk pour une meilleure gestion du cycle de vie inter-comptes. Cependant, le modèle fondamental client vs utilisateur vs planifié reste pertinent dans un avenir prévisible.

- **Communauté et formation :** Enfin, la communauté s'oriente vers une formation plus formelle sur SuiteScript. L'exemple du blog Oracle de fin 2023 (Source: blogs.oracle.com) montre un investissement continu dans l'enseignement des meilleures pratiques. À mesure que la base de clients de NetSuite s'agrandit, davantage de développeurs (souvent non experts en JavaScript) utiliseront SuiteScript. Cela signifie que les directives (comme celles de ce rapport) deviennent encore plus importantes. NetSuite prévoit de continuer à améliorer SuiteScript, mais n'a pas introduit de nouveaux types de scripts fondamentaux au-delà de Map/Reduce récemment, ce qui suggère que ces trois types principaux resteront essentiels.

En résumé, choisir entre les scripts Client, Utilisateur et Planifiés reste une connaissance fondamentale pour les développeurs NetSuite. Les principes décrits ici – quel script s'exécute où et pourquoi – resteront applicables même à mesure que les API évoluent. Les améliorations futures (TypeScript, Redwood, nouvelles API) augmentent principalement la manière dont les scripts sont écrits et gérés, sans pour autant renverser les distinctions de base couvertes dans ce rapport.

Conclusion

Les scripts **Client**, **User Event** et **Scheduled** de SuiteScript occupent chacun un rôle distinct dans la personnalisation de NetSuite. Les scripts client permettent des comportements d'interface immédiats et réactifs dans le navigateur (Source: docs.oracle.com), ce qui les rend essentiels pour la validation des entrées utilisateur et les interactions dynamiques avec les formulaires. Les scripts User Event garantissent que chaque enregistrement sauvegardé ou modifié sur le serveur respecte les règles métier (Source: docs.oracle.com) ; ils fournissent un filet de sécurité pour l'intégrité des données et automatisent les actions associées. Les scripts planifiés prennent en charge les tâches lourdes d'automatisation périodique, permettant aux mises à jour de données à grande échelle, à la maintenance et aux tâches d'intégration de s'exécuter discrètement en arrière-plan (Source: docs.oracle.com) (Source: www.brokenrubik.com).

Le choix entre ces options est guidé par le **moment** et l'**endroit** où votre logique doit s'exécuter. Si elle doit se produire pendant que l'utilisateur tape ou affecter immédiatement le formulaire visible, un script client est approprié. Si elle doit se produire chaque fois qu'un enregistrement est sauvegardé (peu importe la méthode), un script User Event est l'outil adéquat. Si elle peut être différée en dehors des heures de bureau ou exécutée de manière répétée selon un calendrier, utilisez un script planifié. Derrière ces choix se cachent des différences concrètes en termes de ressources système, d'impact sur les performances et de gouvernance, telles que documentées par Oracle (Source: docs.oracle.com) (Source: docs.oracle.com) et expérimentées par les praticiens (Source: www.tvarana.com) (Source: docs.oracle.com).

Nous avons passé en revue la manière dont ces types de scripts sont définis, leurs points d'entrée et les meilleures pratiques (soutenues par la documentation officielle et les conseils d'experts (Source: suiterep.com) (Source: docs.oracle.com). Nous avons illustré leur utilisation avec des exemples d'experts et des modèles de code (Source: www.thenetsuitepro.com) (Source: blogs.oracle.com). Tout au long de ce rapport, il a été démontré qu'une mauvaise utilisation (trop de scripts, scripts UE longs, ou scripts client sur des PC lents) peut dégrader les performances ou dépasser les limites, tandis qu'une utilisation correcte automatise considérablement les flux de travail.

Pour l'avenir, le développement SuiteScript continuera de mettre l'accent sur la robustesse (via TypeScript et SDF) et les pratiques adaptées au cloud (décharger le travail lourd en arrière-plan, surveiller l'utilisation). À mesure que l'interface utilisateur de NetSuite évolue (par exemple Redwood), les rôles fondamentaux du code côté client par rapport au côté serveur s'adapteront sans pour autant disparaître. En fin de compte, comprendre les distinctions et les forces des scripts Client, User Event et Scheduled est essentiel pour tout projet de personnalisation NetSuite, garantissant des solutions efficaces, maintenables et alignées sur l'architecture de NetSuite (Source: docs.oracle.com) (Source: docs.oracle.com).

Toutes les affirmations et conseils contenus dans ce document sont étayés par la documentation officielle de NetSuite et les ressources de praticiens experts (des références ont été fournies). Les développeurs et les architectes doivent consulter les sources citées pour plus de détails, et continuer à tester et profiler leurs scripts spécifiques au fur et à mesure qu'ils implémentent des solutions personnalisées.

Étiquettes: netsuite-suitescript, script-client, script-user-event, script-planifie, suitescript-2x, suitecloud, developpement-netsuite, limites-de-gouvernance

---

AVERTISSEMENT

Ce document est fourni à titre informatif uniquement. Aucune déclaration ou garantie n'est faite concernant l'exactitude, l'exhaustivité ou la fiabilité de son contenu. Toute utilisation de ces informations est à vos propres risques. Houseblend ne sera pas responsable des dommages découlant de l'utilisation de ce document. Ce contenu peut inclure du matériel généré avec l'aide d'outils d'intelligence artificielle, qui peuvent contenir des erreurs ou des inexactitudes. Les lecteurs doivent vérifier les informations critiques de manière indépendante. Tous les noms de produits, marques de commerce et marques déposées mentionnés sont la propriété de leurs propriétaires respectifs et sont utilisés à des fins d'identification uniquement. L'utilisation de ces noms n'implique pas l'approbation. Ce document ne constitue pas un conseil professionnel ou juridique. Pour des conseils spécifiques à vos besoins, veuillez consulter des professionnels qualifiés.