

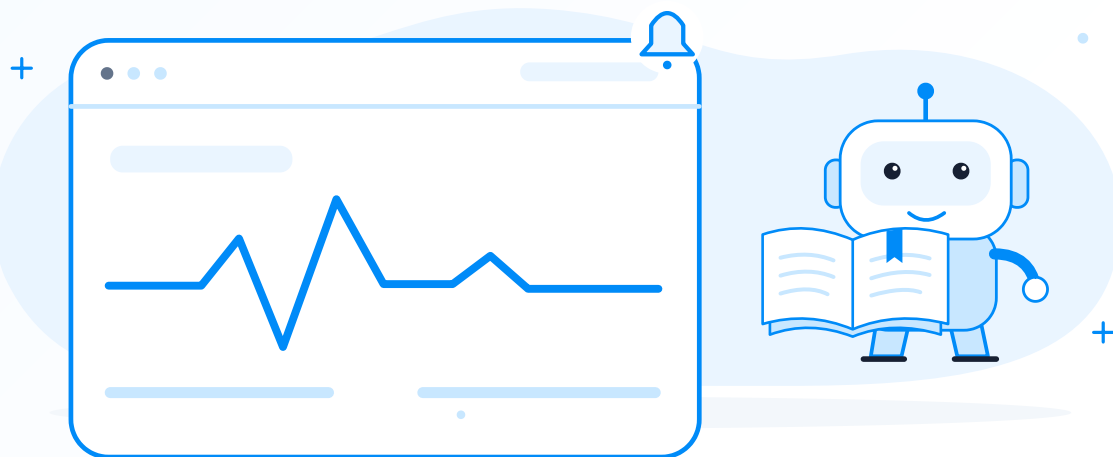
Mini-série

Agents · MCP · SRE / MLOps

Construire un copilote de réponse aux incidents avec MCP

Construire un copilote qui rassemble le contexte d'un incident de production à partir de services, alertes, déploiements et runbooks, sans jamais modifier l'infrastructure.

Auteur **Youness Boutyour**

**DURÉE**

4 épisodes · 45 min /
épisode

NIVEAU

Intermédiaire

FORMAT

Mini-série · 4 épisodes

Agents · MCP · SRE / MLOps · publié le 2026-04-03 · hybrid

Objectifs et livrables

Une alerte vient de signaler une hausse de latence sur `payments-api`, juste après un déploiement. Pour l'astreinte, il faut rapidement retrouver le service concerné, vérifier les changements récents et consulter le bon runbook. Ce tutoriel propose de réunir ces informations dans un copilote de réponse aux incidents.

Nous allons préparer un jeu de données synthétiques qui relie services, alertes, déploiements et procédures, puis exposer quatre outils MCP en lecture seule. Le copilote pourra rechercher un service, consulter son propriétaire et ses signaux, lister ses déploiements et retrouver les étapes d'un runbook. Vous inspecterez ensuite ces outils avant de vous en servir pour rédiger une fiche d'incident.

Cette fiche réunira les observations disponibles, les hypothèses à examiner et les actions soumises à validation humaine. Chaque appel sera conservé dans `logs/mcp-calls.log` : vous pourrez ainsi retrouver l'origine d'une information et comprendre pourquoi une étape a été proposée.

Le périmètre reste celui de l'aide au diagnostic. Les redémarrages, changements de configuration et roll-backs restent à la charge de l'ingénieur. Nous examinerons aussi ce que le copilote doit répondre lorsqu'une information manque, afin que la fiche reflète les données réellement consultées.

RÉSULTATS DES ÉTAPES · MCP

01 · ENVIRONNEMENT

```
$ python --version
Python 3.11.x

$ py_compile src/server.py
✓ terminé sans erreur

requirements.lock.txt
```

02 · INSPECTOR

```
tools/list
find_service
get_service_context
get_runbook

✓ 4 outils en lecture
aucune écriture exposée
```

03 · FICHE

```
service: payments-api
deployment: rc3
scope: read-only

✓ logs/mcp-calls.log

validation humaine requise
```

Progression de réalisation du copilote, de la donnée synthétique à la fiche d'incident.

FICHE D'INCIDENT · SORTIE DE RÉFÉRENCE

payments-api · latence élevée après déploiement

OBSERVATIONS

Service **payments-api**
Déploiement **2026.09.18-rc3**
Signaux **latency · error-rate**

PÉRIMÈTRE

- ✓ lecture seule
- ✓ appels journalisés
- × aucun rollback

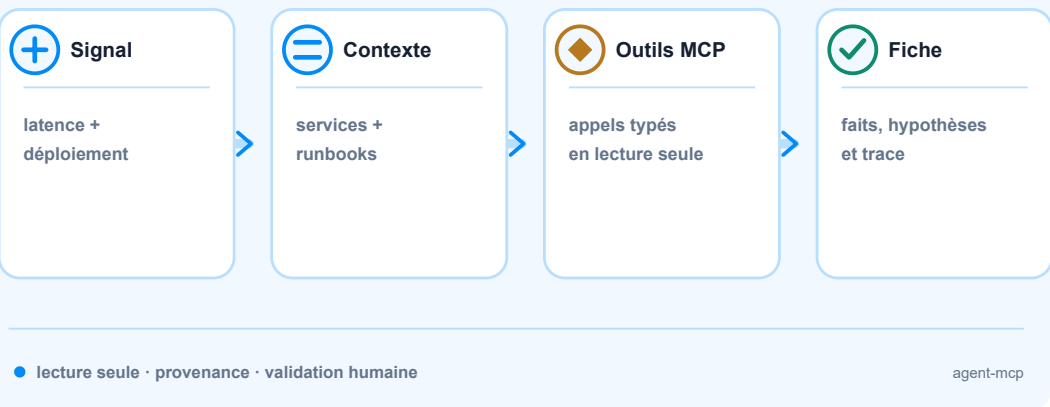
PREUVE À CONSERVER

logs/mcp-calls.log · réponse tools/call · validation humaine avant action
Exemple de sortie reproductible — les données restent synthétiques.

Exemple de résultat final et de trace vérifiable.

Du signal incident à une fiche traçable

Schéma du copilote MCP qui rassemble un signal, des sources opérationnelles, des outils en lecture seule et une fiche validable.



Le raisonnement suit quatre frontières : signal, contexte, outils en lecture seule et fiche d'incident.

Lire le flux avant de coder

Le schéma sépare volontairement le signal de la fiche finale. Une alerte indique ce qui a été observé ; elle ne constitue pas encore une cause. Les outils MCP servent à compléter ce contexte avec des données identifiables, tandis que la fiche distingue les faits, les hypothèses et les actions qui demandent une validation humaine.

À chaque appel, vérifiez trois éléments : l'identifiant demandé, la source réellement consultée et la présence d'une trace dans logs/mcp-calls.log. Si une information manque, le résultat correct est une fiche incomplète mais honnête, pas une valeur inventée. Cette règle explique pourquoi les outils restent en lec-

ture seule dans la démonstration.

1. Préparer l'environnement

Préparez un dossier de travail qui contiendra le serveur, les données d'incident et les journaux. L'environnement virtuel doit rester actif dans le terminal qui lancera le serveur ou Inspector ; il détermine quel interpréteur Python et quelle version du SDK sont utilisés. Le contrôle de Node.js concerne Inspector, tandis que le serveur métier reste un programme Python.

Conservez les versions effectivement installées avec les fichiers du projet. Le gel des dépendances facilite la comparaison entre deux essais, mais ne démarre aucun serveur : cette étape vérifie seulement que les outils nécessaires sont disponibles. Les premières vérifications du contenu métier se feront ensuite sur le fichier JSON, avant d'ajouter la communication MCP.

```
Terminal · Bash

mkdir -p incident-copilot/{data,src,logs}
cd incident-copilot
python3.11 -m venv .venv
source .venv/bin/activate
python -m pip install --upgrade pip
python -m pip install --upgrade "mcp ≥ 2, < 3"
python -m pip freeze > requirements.lock.txt
node --version
```

Sortie obtenue

Sortie / exemple

```
Python 3.11.x
v22.x.x
```

Commande de vérification

Conservez `requirements.lock.txt`, la sortie des deux commandes de version et l'absence d'erreur à l'installation. Si `python3.11` ou `node` n'est pas disponible, arrêtez-vous ici et corrigez l'environnement avant de continuer.

Conservez `requirements.lock.txt`, la version de Python et la version de Node utilisée par Inspector. Le serveur sera local et lancé avec le transport standard utilisé par l'Inspector. La plage `mcp ≥ 2, < 3` suit la version majeure actuelle tout en évitant une rupture automatique lors d'une future version 3 ; régénérez le fichier de verrouillage lorsque vous actualisez l'environnement.

2. Créer un contexte d'incident synthétique mais réaliste

Le fichier `operations.json` rassemble un petit contexte d'astreinte : services, responsables, alertes, déploiements et procédures. Les noms de services servent à relier ces éléments ; gardez-les cohérents plutôt que de créer plusieurs variantes pour le même service. Lisez un incident de bout en bout dans ce fichier pour comprendre quelles informations chaque outil devra restituer.

Les alertes donnent des observations, pas un diagnostic. Une hausse de latence et un déploiement proche dans le temps justifient une vérification, sans démontrer un lien causal. Comparez aussi les unités et les fenêtres de mesure avant d'opposer deux valeurs. Ce jeu de données permet justement d'écrire une fiche qui distingue les faits disponibles des hypothèses et des contrôles encore à réaliser.

Créez `data/operations.json` :

```
JSON

{
  "services": [
    {
      "name": "payments-api",
      "owner": "platform-payments",
      "criticality": "high",
      "alerts": [
        {"id": "INC-2048", "type": "latency", "severity": "high", "value": "1.8 s", "window": "10 min"},
        {"id": "INC-2049", "type": "error-rate", "severity": "medium", "value": "3.2 %", "window": "10 min"}
      ],
      "deployments": [
        {"version": "2026.09.18-rc3", "environment": "production", "deployed_at": "2026-09-18T08:42:00Z", "status": "success"},
        {"version": "2026.09.11-rc8", "environment": "production", "deployed_at": "2026-09-11T14:20:00Z", "status": "success"}
      ],
      "runbooks": {
        "latency": [
          "Vérifier la corrélation avec le dernier déploiement.",
          "Comparer p95 et taux d'erreur avec la fenêtre précédente.",
          "Demander une validation humaine avant toute action de mitigation."
        ],
        "error-rate": [
          "Identifier les routes et dépendances les plus touchées.",
          "Comparer les erreurs avec les logs applicatifs autorisés."
        ]
      }
    },
    {
      "name": "identity-api",
      "owner": "platform-identity",
      "criticality": "medium",
      "alerts": [
        {"id": "INC-2071", "type": "latency", "severity": "medium", "value": "820 ms", "window": "15 min"}
      ]
    }
  ]
}
```

```
{
  "id": "INC-2072", "type": "error-rate", "severity": "low", "value": "1.1 %", "window": "15 min"},
  "deployments": [
    {
      "version": "2026.09.19-rc2", "environment": "production", "deployed_at": "2026-09-19T10:05:00Z",
      "status": "success"
    },
    {
      "version": "2026.09.12-rc5", "environment": "production", "deployed_at": "2026-09-12T16:30:00Z",
      "status": "success"
    }
  ],
  "runbooks": {
    "latency": [
      "Comparer p95 entre les routes de connexion et de renouvellement.",
      "Vérifier la disponibilité du fournisseur d'identité.",
      "Demander une validation humaine avant toute action de mitigation."
    ],
    "error-rate": [
      "Identifier les clients et routes les plus touchés.",
      "Comparer les erreurs avec les journaux applicatifs autorisés."
    ]
  }
}
```

Ce fichier joue le rôle d'un adaptateur vers des systèmes réels : catalogue de services, fournisseur d'alertes, registre de déploiements et base de runbooks. En production, ces sources seraient interrogées séparément avec des droits minimaux. Aucune clé, aucun token et aucune donnée personnelle ne doit apparaître dans le jeu de test.

Sortie obtenue

Validez le JSON avant de lancer le serveur :

Terminal · Bash

```
python -m json.tool data/operations.json >/dev/null && echo "catalogue JSON: PASS"
```

Sortie / exemple

```
catalogue JSON: PASS
```

Commande de vérification

Le fichier doit contenir `payments-api`, `identity-api`, les alertes des deux services, les déploiements récents et les runbooks `latency` et `error-rate`. Cette vérification évite de diagnostiquer un problème du MCP alors que le jeu de données est invalide.

3. Implémenter le serveur MCP

Le serveur charge un fichier situé relativement à son propre emplacement, plutôt qu'un chemin fourni par l'agent. Les fonctions exposent ensuite des lectures ciblées de ces données. Commencez par rechercher le service, puis réutilisez son nom précis pour les appels suivants : une recherche large par symptôme ou par responsable peut correspondre à plusieurs entrées.

Les petits détails de sélection influencent la fiche finale. La liste de déploiements conserve l'ordre du fichier : pour la lire comme une chronologie récente, vérifiez que les données sont déjà rangées dans cet ordre. Une procédure absente renvoie un statut explicite, sans en inventer une. Les journaux vont dans un fichier afin de laisser la sortie standard disponible pour le transport stdio.

Créez `src/server.py` :

```
Python

from __future__ import annotations

import json
import logging
from pathlib import Path

from mcp.server.mcpserver import MCPServer

ROOT = Path(__file__).resolve().parents[1]
DATA_FILE = ROOT / "data" / "operations.json"
LOG_FILE = ROOT / "logs" / "mcp-calls.log"
LOG_FILE.parent.mkdir(exist_ok=True)

logging.basicConfig(
    filename=LOG_FILE,
    level=logging.INFO,
    format="%asctime)s %(levelname)s %(message)s",
)

mcp = MCPServer("incident-response-copilot")

def load_operations() → dict:
    return json.loads(DATA_FILE.read_text(encoding="utf-8"))

def find_service_record(name: str) → dict | None:
    wanted = name.casefold().strip()
    if not wanted:
        return None
    for service in load_operations()["services"]:
        haystack = " ".join([
            service["name"],
```

```

        service["owner"],
        service["criticality"],
        *(alert["type"] for alert in service["alerts"]),
    ]).casefold()
    if wanted in haystack:
        return service
    return None

```

```
@mcp.tool()
```

```

def find_service(query: str) → list[dict[str, str]]:
    """Find services matching a name, owner, criticality or alert type."""
    normalized = query.casefold().strip()
    if not normalized:
        logging.warning("tool=find_service query=empty results=0")
        return []
    results = []
    for service in load_operations()["services"]:
        haystack = f"{service['name']} {service['owner']} {service['criticality']} " \
            + " ".join(alert["type"] for alert in service["alerts"])
        if normalized in haystack.casefold():
            results.append({
                "service": service["name"],
                "owner": service["owner"],
                "criticality": service["criticality"],
            })
    logging.info("tool=find_service query=%r results=%d", query, len(results))
    return results

```

```
@mcp.tool()
```

```

def get_service_context(service: str) → dict:
    """Return the non-secret operational context of one service."""
    record = find_service_record(service)
    if record is None:
        logging.warning("tool=get_service_context service=%s result=not_found", service)
        return {"status": "not_found", "service": service}
    result = {
        "service": record["name"],
        "owner": record["owner"],
        "criticality": record["criticality"],
        "alerts": record["alerts"],
    }
    logging.info("tool=get_service_context service=%s result=found", service)
    return result

```

```
@mcp.tool()
```

```

def list_recent_deployments(service: str, limit: int = 3) → list[dict[str, str]]:
    """List recent production deployments without exposing credentials."""
    record = find_service_record(service)
    if record is None:
        logging.warning("tool=list_recent_deployments service=%s result=not_found", service)
        return []

```

```
safe_limit = max(1, min(limit, 10))
result = [item for item in record["deployments"] if item["environment"] == "production"][:safe_limit]
logging.info("tool=list_recent_deployments service=%s results=%d", service, len(result))
return result

@mcp.tool()
def get_runbook(service: str, symptom: str) → dict[str, str | list[str]]:
    """Return read-only runbook steps for a service and an observed symptom."""
    record = find_service_record(service)
    if record is None:
        logging.warning("tool=get_runbook service=%s result=not_found", service)
        return {"status": "not_found", "service": service, "steps": []}
    key = symptom.casefold().strip()
    steps = record["runbooks"].get(key, [])
    logging.info("tool=get_runbook service=%s symptom=%s steps=%d", service, key, len(steps))
    return {"status": "found" if steps else "not_found", "service": service, "symptom": key, "steps": steps}

if __name__ == "__main__":
    mcp.run()
```

Le serveur ne construit aucun chemin à partir des entrées utilisateur, ne lit qu'un fichier autorisé et ne propose aucune fonction d'écriture. Les appels et les résultats sont journalisés sans secret.

Vérifiez le fichier avant de connecter un agent :

```
Terminal · Bash

python -m py_compile src/server.py
```

Sortie obtenue

■ Sortie / exemple

La commande se termine sans sortie et avec le code 0.

Commande de vérification

Le fichier `src/server.py` est compilable et `logs/mcp-calls.log` existe après le premier appel. Une erreur d'import ou de syntaxe doit être corrigée avant l'inspection avec Inspector.

4. Inspecter le copilote avec MCP Inspector

Inspector joue le rôle de client MCP : il lance le processus Python, découvre ses outils et permet de leur envoyer des arguments. Vérifiez d'abord les noms et les schémas annoncés, puis appelez les outils un par un. Cette séparation aide à distinguer un problème de connexion d'une erreur dans une fonction métier.

Pour le même service, comparez les réponses à `operations.json` : propriétaire, alertes, déploiements et procédure doivent correspondre aux entrées du fichier. Testez aussi un nom inconnu pour observer le refus ou l'absence de résultat. Si aucun outil n'apparaît, examinez l'interpréteur et les journaux avant de modifier les données ; un message de diagnostic imprimé sur stdout peut perturber les échanges stdio.

Lancez l'interface dans un second terminal :

Terminal · Bash

```
npx -y @modelcontextprotocol/inspector python src/server.py
```

Dans l'Inspector, vérifiez puis appelez :

1. `find_service` avec `latency` ;
2. `get_service_context` avec `payments-api` ;
3. `list_recent_deployments` avec `payments-api` et `limit=2` ;
4. `get_runbook` avec `payments-api` et `latency` ;
5. `get_runbook` avec `payments-api` et `restart` pour vérifier l'absence d'action d'écriture.

La vérification automatisable des outils est possible avec le client CLI :

Terminal · Bash

```
npx -y @modelcontextprotocol/inspector --cli \
python src/server.py \
--method tools/list \
--format json

npx -y @modelcontextprotocol/inspector --cli \
python src/server.py \
--method tools/call \
--tool-name get_service_context \
--tool-arg service=payments-api \
--format json
```

L'Inspector permet de tester la liste des outils, leurs schémas, leurs appels et les journaux du serveur. Consultez la [documentation officielle de MCP Inspector](#).

Sortie obtenue

■ Sortie / exemple

```
tools/list → find_service, get_service_context,  
            list_recent_deployments, get_runbook  
get_service_context(payments-api) → status: found
```

Commande de vérification

Capturez l'écran de MCP Inspector montrant la liste des outils et un appel réussi. Conservez également la ligne correspondante dans `logs/mcp-calls.log`. L'écran doit permettre de vérifier qu'aucune opération d'écriture n'est exposée.

5. Produire une fiche d'incident

Construisez la fiche à partir des résultats déjà inspectés : le contexte identifie le service, les alertes décrivent l'observation et les déploiements donnent des éléments de chronologie. La procédure fournit des vérifications possibles, pas la preuve qu'elles ont été exécutées. Reprenez les valeurs utiles avec leur fenêtre temporelle plutôt que de produire une synthèse détachée des données.

La consigne proposée guide la rédaction ; elle ne constitue pas à elle seule un client d'agent automatisé. Que la synthèse soit rédigée manuellement ou par un agent connecté, vérifiez chaque affirmation contre les réponses des outils. Une bonne fiche peut conclure que la cause reste inconnue, tout en indiquant précisément quel contrôle effectuer ensuite et quel responsable contacter.

Une consigne d'agent peut être :

■ Sortie / exemple

Tu es le copilote de réponse aux incidents de l'équipe plateforme.

Objectif : produire une fiche factuelle pour l'incident transmis.

Utilise uniquement les outils MCP disponibles en lecture seule.

Pour chaque constat, indique l'outil utilisé et la donnée observée.

Commence par identifier le service, puis récupère son contexte, ses déploiements récents et le runbook du symptôme.

Ne redémarre rien, ne modifie aucune configuration et ne propose pas de rollback automatique.

Si une information est absente, écris « information indisponible ».

Format de sortie :

1. Service et criticité
2. Alertes observées
3. Dernier déploiement connu
4. Hypothèses à vérifier

5. Étapes de runbook autorisées
6. Actions explicitement hors périmètre
7. Outils et sources consultés

Pour le scénario `payments-api`, la fiche doit relier l'alerte de latence au dernier déploiement `2026.09.18-rc3`, mentionner le taux d'erreur associé et proposer uniquement les vérifications prévues par le runbook. Elle ne doit pas déclarer que le déploiement est la cause sans élément vérifiable.

Sortie obtenue

■ Sortie / exemple

```
Service      : payments-api
Déploiement  : 2026.09.18-rc3
Alertes      : latency=1.8 s, error-rate=3.2 %
Action       : validation humaine requise ; aucun rollback automatique
```

Commande de vérification

Comparez la fiche avec les réponses JSON réellement retournées par les quatre outils. La fiche est incorrecte si elle transforme une corrélation temporelle en cause prouvée ou si elle propose une action d'écriture.

Exemples de sorties

Après l'inspection, conservez les sorties suivantes :

■ Sortie / exemple

```
tools/list      → find_service, get_service_context,
                  list_recent_deployments, get_runbook
get_service_context → payments-api · critical · owner=platform
list_recent_deployments → 2026.09.18-rc3
get_runbook(latency) → vérifications autorisées, aucune écriture
```

Le contrôle minimal combine la réponse JSON, `logs/mcp-calls.log` et la fiche d'incident. Une sortie valide ne doit contenir ni secret, ni chemin construit à partir d'une entrée utilisateur, ni commande de rollback.

Dépannage et diagnostics

Symptôme	Cause probable	Vérification
<code>ModuleNotFoundError</code> pour <code>mcp</code>	environnement virtuel non activé ou dépendance absente	<code>python -m pip show mcp</code> puis <code>python -m py_compile src/server.py</code>
aucun outil dans Inspector	serveur non lancé ou protocole mal sélectionné	relancer <code>npx -y @modelcontextprotocol/inspector python src/server.py</code>
réponse <code>not_found</code>	service ou symptôme absent du catalogue	vérifier <code>data/operations.json</code> , sans élargir arbitrairement le périmètre
log contenant une donnée sensible	journalisation trop détaillée	supprimer la valeur, conserver le nom de l'outil et rejouer le test

Vérification finale — sécurité et périmètre

Ajoutez ces tests au rapport :

Test	Sortie obtenue
recherche <code>latency</code>	<code>payments-api</code> est trouvé
service inconnu	résultat <code>not_found</code> , sans exception brute
limite <code>1000</code>	au maximum 10 déploiements sont retournés
symptôme <code>restart</code>	aucun outil d'écriture ni commande d'action n'est proposé
requête <code>../secrets</code>	aucune construction de chemin, aucune lecture hors catalogue
inspection du journal	outils et arguments sont traçables, aucun secret présent
séance ou service brouillon ajouté au catalogue	l'agent ne doit pas l'inventer

Dans une vraie intégration, ajoutez une authentification du transport, des permissions par outil, une rétention des logs et une revue humaine avant toute action. Le principe important est la séparation entre **observer**, **recommander** et **agir**.

Limites et prolongements

Le catalogue JSON est un adaptateur local. Pour une architecture de production, remplacez-le par des connecteurs séparés vers l'observabilité, le registre de déploiement, la base de runbooks et le catalogue de services. Chaque connecteur doit exposer le minimum de données nécessaire.

Vous pouvez ensuite ajouter un outil de corrélation de fenêtres temporelles, une ressource de runbook versionnée ou un prompt MCP dédié. Ajoutez ces capacités une par une, avec un test de schéma, un cas d'erreur et une trace dans le journal. Le copilote devient intéressant lorsqu'il raccourcit le temps de compréhension d'un incident sans devenir une porte d'entrée non contrôlée vers la production.

Références

- [Model Context Protocol — documentation officielle](#)
- [MCP Inspector — documentation et code source](#)