Développer un serveur MCP en Python

Concevoir, tester et déployer un serveur MCP prêt pour la production avec MCP Inspector, mcp.json et Streamable HTTP

Ayi NEDJIMI

Développer un serveur MCP en Python

Concevoir, tester et déployer un serveur MCP prêt pour la production avec MCP Inspector, mcp.json et Streamable HTTP

Ayi Nedjimi

Ayi Nedjimi Consultants

Développer un serveur MCP en Python
Première Édition

Première publication : 2026

ISBN 978-2-XXXXX-XXX-X

Publié par Ayi Nedjimi Consultants
ayinedjimi-consultants.fr

Table des matières

À propos de l'auteur

Ayi NEDJIMI

Auditeur senior en cybersécurité et consultant en intelligence artificielle

Ayi NEDJIMI est auditeur senior en cybersécurité et consultant en intelligence artificielle. Fort de plus de 25 ans d'expérience sur des missions critiques, il accompagne les organisations dans l'évaluation de leurs risques, la sécurité de leurs infrastructures et leurs projets d'IA.

Ancien développeur Microsoft à Redmond, il a travaillé sur le module d'authentification GINA de Windows NT4. Il est également co-auteur de la version française du guide de sécurité Windows NT4 pour la NSA.

À la tête d'Ayi NEDJIMI Consultants, il intervient sur des audits ISO 42001 et ISO 27001, des tests d'intrusion d'infrastructures critiques, des investigations numériques et des missions de conformité NIS2 et AI Act. Ses domaines d'expertise couvrent aussi les environnements cloud, Active Directory, les grands modèles de langage et les architectures RAG.

Expert judiciaire auprès de la Cour d'appel de Paris, il dispose d'une habilitation Confidentiel Défense.

Conférencier en Europe et aux États-Unis, il a formé plus de 10 000 professionnels. Son parcours comprend plus de 100 missions et la publication de plus de 700 articles techniques. Il partage son expertise à travers la formation, ses publications et ses projets consacrés à la cybersécurité et à l'intelligence artificielle.

Retrouver l'auteur

ayinedjimi-consultants.fr · ayi@ayinedjimi-consultants.fr

linkedin.com/in/ayi-nedjimi · github.com/ayinedjimi · orcid.org/0009-0001-5056-3253

Introduction

Au fil de ce livre, vous allez créer, tester et déployer un serveur MCP de référence en Python, conçu pour préparer une mise en production. Il proposera deux outils, add et joke, une ressource et un modèle de prompt. Vous apprendrez à l'intégrer à des applications hôtes à l'aide de mcp.json, à l'utiliser depuis votre propre client, puis à passer au transport Streamable HTTP en ajoutant des protections adaptées. Enfin, vous intégrerez MCP Inspector à votre pipeline d'intégration continue (CI) pour détecter les régressions.

Ce livre s'adresse à vous si vous savez exécuter du Python, connaissez les outils de développement courants et pouvez installer Node.js pour utiliser Inspector. Il vous aidera à fournir un serveur MCP utilisable dans un éditeur comme dans votre propre application, avec des mesures de sécurité et des contrôles automatisés adaptés. Pour suivre les exercices, vous devez disposer d'une application hôte capable d'utiliser un grand modèle de langage (LLM) et pouvoir exécuter des outils de développement en local.

1

Comprendre MCP et définir votre serveur

MCP fournit une manière standard de mettre des fonctionnalités à la disposition des LLM. Vous évitez ainsi de recréer sans cesse des intégrations spécifiques. Voyez-y une extension de l'appel de fonctions : une interface commune permet au modèle de découvrir et d'utiliser des outils, des ressources et des prompts de façon fiable. Vous pouvez alors consacrer davantage de temps à votre application.

Trois difficultés expliquent l'intérêt de MCP. La première concerne les limites et la fragmentation du contexte. Les prompts et les réponses doivent tenir dans la fenêtre de contexte du modèle. Plus une conversation s'allonge, plus son coût augmente ; renvoyer tout l'historique à chaque échange peut aussi nuire à la pertinence des réponses. Il faut donc récupérer le contexte utile, sans reprendre systématiquement l'ensemble de la discussion. La deuxième difficulté est l'intégration des outils et de la mémoire. Les architectures d'agents combinent souvent plugins, API privées et schémas conçus au cas par cas. Reproduire ces assemblages d'une application à l'autre coûte cher et les rend fragiles. MCP standardise la découverte des capacités d'un serveur et la manière de les appeler. La troisième difficulté est l'assemblage de services. Une application peut avoir besoin de calendriers, de fichiers,

de paiements et de recherches, chacun utilisant ses propres formats. MCP permet de répartir ces domaines entre plusieurs serveurs qui présentent une interface commune au client.

Tableau 1.1 — Un standard commun réduit les coûts d'intégration et permet de concevoir des processus plus riches avec des agents.

Sans interface commune	Avec un standard commun
Les limites actuelles conduisent à	Un standard ouvre de nouvelles possibilités
Créer des intégrations coûteuses pour partager des fonctionnalités et ajouter des plugins.	Assembler facilement des agents à partir de plusieurs serveurs.
Limiter les conversations à cause de la fenêtre de contexte, au détriment de l'expérience utilisateur.	Créer de nouveaux parcours : utiliser le serveur X pour réserver un voyage et le serveur Y pour réaliser des opérations bancaires.
Limiter les conversations à cause de la fenêtre de contexte, au détriment de l'expérience utilisateur.	Atténuer plusieurs difficultés liées aux fenêtres de contexte limitées.

Les bénéfices sont concrets : des intégrations plus rapides, des capacités plus faciles à découvrir et moins de complexité inutile. Vous pouvez décomposer une application monolithique, conserver des frontières claires entre les domaines et offrir au LLM un point d'accès cohérent aux fonctionnalités. Des descriptions précises réduisent les erreurs de sélection des outils, tandis que des ressources ciblées limitent le contexte superflu. Vous passez aussi moins de temps à débattre des SDK : le protocole reste commun, même lorsque vos services internes évoluent.

Trois concepts fondamentaux

MCP repose sur trois éléments que vous utiliserez régulièrement : les outils, les ressources et les prompts.

- Les outils exécutent des opérations. Ils reçoivent des paramètres et effectuent un traitement.
- Les ressources fournissent du contexte statique. Le client les lit au lieu de les exécuter.
- Les prompts sont des modèles d'instructions réutilisables. Ils formalisent une approche éprouvée pour éviter de repartir de zéro à chaque demande.

Concevez vos outils comme des fonctions d'API dont le contrat est explicite. Donnez aux paramètres des noms clairs et des types précis. Rédigez une description en anglais simple que le LLM pourra rapprocher des demandes en langage naturel. Cette description compte davantage que le seul nom de la fonction : elle relie les mots de l'utilisateur au fonctionnement du serveur. Les ressources apportent des informations, par exemple un fichier de configuration, une courte spécification ou un petit schéma JSON. Limitez-les à ce qui est pertinent pour préserver la fenêtre de contexte. Quant aux prompts, proposez des modèles éprouvés, avec des variables, plutôt que de longues instructions difficiles à réutiliser.

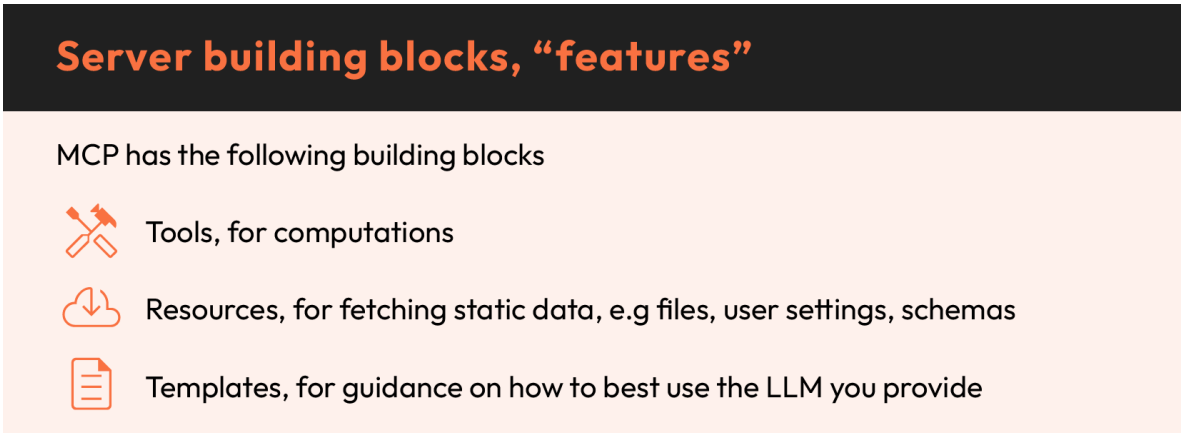

Figure 1.1 : Outils, ressources et modèles de prompt se complètent pour créer des agents capables d'agir à partir d'un contexte fiable.

Libellés de la figure : les composants du serveur sont les outils, pour effectuer des traitements ; les ressources, pour récupérer des données statiques telles que des fichiers, des paramètres utilisateur et des schémas ; les modèles de prompt, pour guider l'utilisation du LLM.

Prenez le temps de bien définir ces éléments. Des schémas de paramètres imprécis et des descriptions vagues provoquent des appels inadaptés. Des ressources trop volumineuses augmentent les coûts et dégradent les réponses. Des prompts mal conçus noient les informations utiles. Soyez précis dans les descriptions, fournissez un contexte minimal mais suffisant et utilisez les prompts pour réemployer vos meilleures instructions. C'est ainsi que vous obtiendrez un comportement prévisible malgré une fenêtre de contexte limitée.

Vue d'ensemble de l'architecture

Retenez trois rôles : l'hôte, le client et le serveur. L'hôte porte l'expérience utilisateur de l'agent, par exemple une conversation intégrée à un éditeur. Le client communique en MCP et maintient une session dédiée avec un serveur. Le serveur expose les outils, les ressources et les prompts, puis interroge vos sources de données. Les échanges prennent la forme de messages JSON-RPC : requêtes, résultats et, lorsque nécessaire, erreurs.

Le client peut intégrer un LLM ou rester plus simple. Avec un LLM, il peut interpréter une demande en langage naturel et la faire correspondre à un outil. Sans modèle, il peut tout de même appeler directement les outils, mais cette interprétation automatique disparaît. Dans les deux cas, chaque session relie un client à un serveur : le client découvre les outils, les ressources et les prompts disponibles, puis les utilise.

Lecture de la figure : les hôtes sont des applications utilisant un LLM, comme Claude Desktop ou un environnement de développement. Ils établissent les connexions. Dans l'hôte, chaque client MCP maintient une connexion dédiée à un serveur. La couche de transport relie les deux. Les serveurs fournissent aux clients du contexte, des outils et des prompts. Host = hôte ; MCP Client = client MCP ;

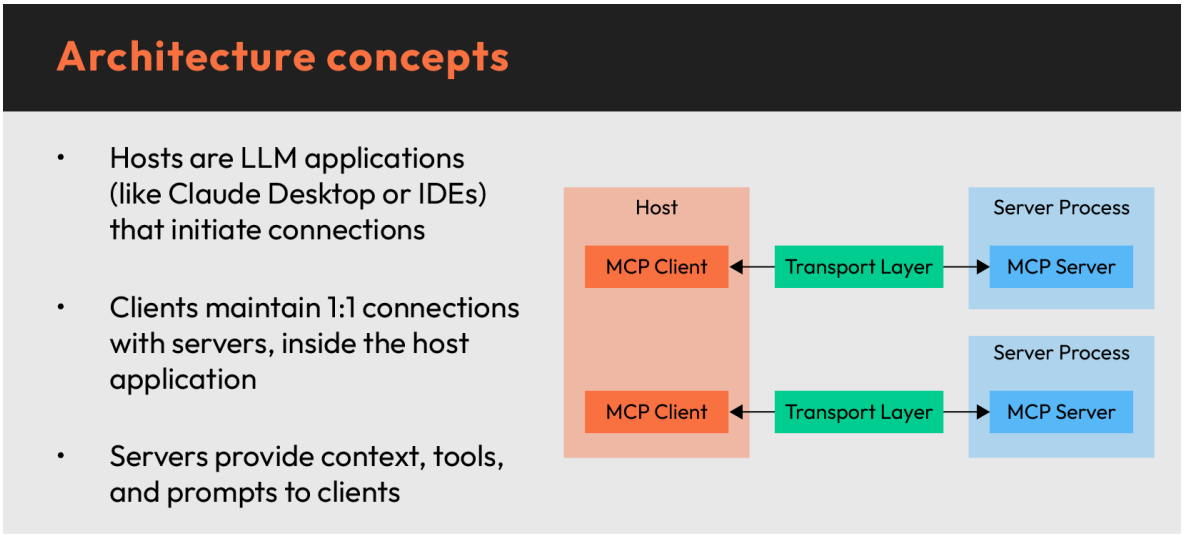

Figure 1.2 : Les hôtes exécutent les clients. Chaque session relie un client à un serveur par une couche de transport.

Transport Layer = couche de transport ; Server Process = processus serveur ; MCP Server = serveur MCP.

Cette séparation facilite l'évolution de l'ensemble. Les hôtes peuvent changer de client ou se connecter à plusieurs serveurs. Les serveurs peuvent passer d'un transport local à un transport distant tout en conservant leur contrat. Les clients découvrent les capacités à la connexion, sans coder en dur chaque point d'accès. Lorsqu'un nouveau serveur est ajouté à un hôte, le client peut ainsi découvrir ses capacités, négocier les fonctionnalités prises en charge et commencer à l'utiliser, sans adaptateur spécifique.

Comprendre l'échange d'initialisation

La communication commence par un échange d'initialisation, souvent appelé handshake : initialize, puis initialized. Le client contacte le serveur et envoie initialize avec les fonctionnalités qu'il prend en charge. Le serveur répond en annonçant ses propres capacités. Le client envoie ensuite la notification initialized pour signaler qu'il est prêt. Les appels ordinaires peuvent alors commencer. Cet échange constitue la négociation des capacités.

Lecture de la figure : le client envoie une requête initialize avec sa version du protocole et ses capacités. Le serveur répond avec sa version et ses capacités. Le client envoie la notification initialized comme accusé de réception, puis les échanges habituels commencent. Initialize request = requête initialize ; Initialize response = réponse à initialize ; Initialized notification = notification initialized ; Connection ready for use = connexion prête à être utilisée.

Sans cet échange, le client doit faire des suppositions sur les fonctionnalités disponibles, ce qui entraîne des erreurs évitables. Après l'initialisation, il sait ce que le serveur propose et comment l'utiliser. Les

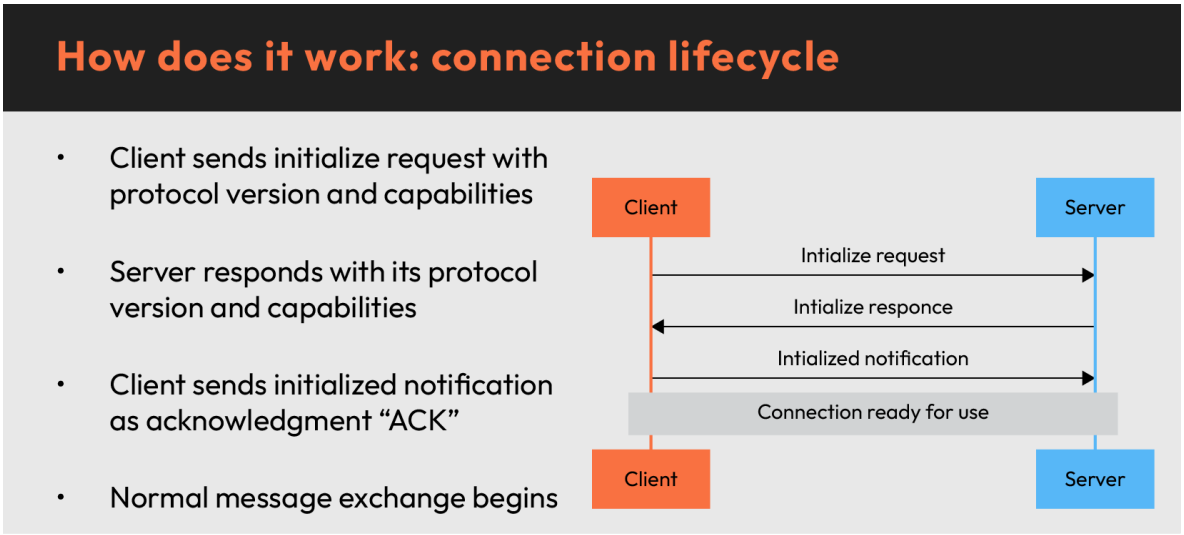

Figure 1.3 : Le client et le serveur échangent les messages initialize et initialized avant de commencer les opérations habituelles.

SDK masquent le format des messages, mais gardez à l'esprit cet ordre : échanger les capacités, puis les utiliser. Cela permet au serveur d'évoluer sans perturber les clients qui respectent le protocole.

Les problèmes apparaissent souvent lorsque les équipes oublient cette étape et codent en dur des hypothèses sur les transports, les noms d'outils ou les fonctions facultatives. Le jour où le serveur ajoute ou retire une capacité, le client ne fonctionne plus. Considérez l'initialisation comme une vérification de compatibilité avant de commencer le travail.

Choisir le transport et le lieu d'exécution

Choisissez le transport dès le départ. Pour le développement local, stdio est le plus simple : le serveur utilise l'entrée et la sortie standard, et le client lance son processus. Vous pouvez ainsi tester rapidement chaque modification. Pour un usage distant, privilégiez Streamable HTTP. Le transport historique HTTP avec SSE est en cours de remplacement ; conservez-le si des clients en dépendent encore, tout en préparant la migration.

Streamable HTTP simplifie l'intégration grâce à un point d'accès unique et à des échanges JSON par défaut. Le transport historique avec SSE utilise souvent des routes distinctes et des contenus textuels qu'il faut ensuite analyser ; ses possibilités sont également plus limitées. Streamable HTTP constitue la voie de migration présentée ici et s'intègre aux pratiques habituelles de déploiement des services web.

Les transports et leurs usages

La couche de transport assure la communication entre les clients et les serveurs. MCP propose plusieurs mécanismes.

Stdio : utilise l'entrée et la sortie standard. Il convient aux processus locaux.

HTTP avec SSE : utilise Server-Sent Events pour les messages du serveur vers le client, et des requêtes HTTP POST dans l'autre sens.

Streamable HTTP : utilise un point d'accès HTTP unique et des messages JSON-RPC. Il convient aux services distants et aux déploiements web avec proxy et authentification.

Commencez avec stdio pour développer et tester en local. Préparez ensuite le serveur à proposer aussi Streamable HTTP lorsque vous serez prêt à le déployer. Si vous utilisez déjà l'ancien transport SSE, préservez temporairement la compatibilité des clients existants et ajoutez un accès Streamable HTTP. Vous réduirez le nombre de composants à configurer et faciliterez l'ajout ultérieur d'une passerelle et de l'authentification.

Quel que soit le transport, les échanges reposent sur les mêmes messages JSON-RPC 2.0. Stdio et Streamable HTTP diffèrent par la façon d'acheminer les données, mais transportent les mêmes requêtes, comme tools/list ou tools/call, et des réponses contenant soit result, soit error.

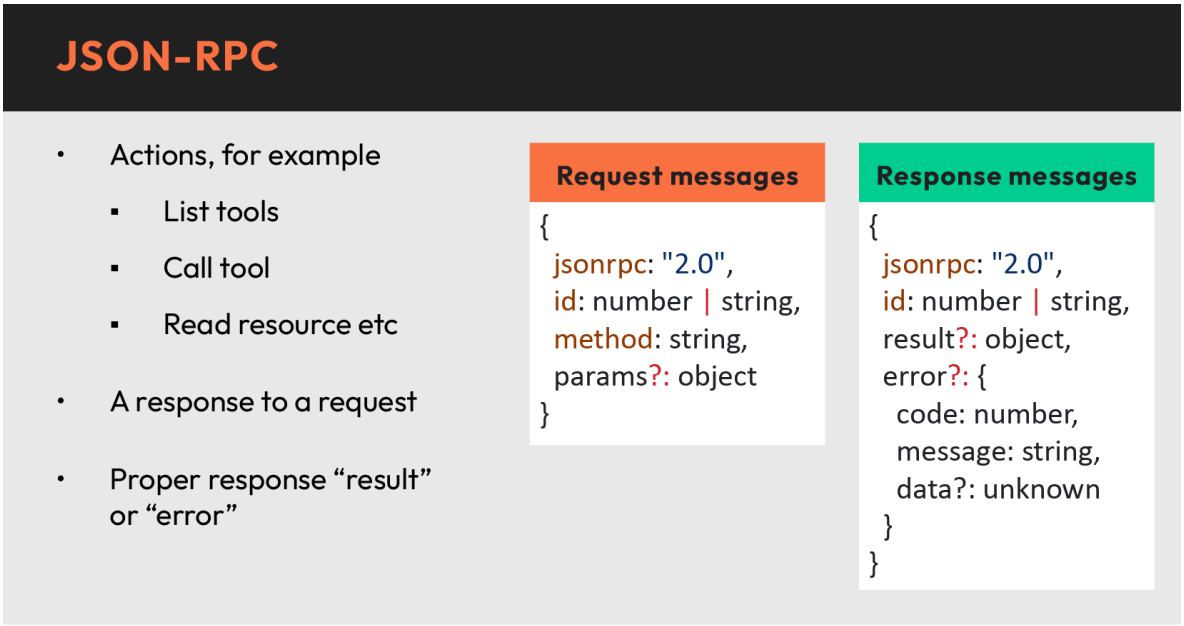

Figure 1.4 : Exemple de requête JSON-RPC, telle que tools/list, et de réponse contenant result en cas de succès ou error en cas d'échec.

Lecture de la figure : les actions comprennent la liste des outils, l'appel d'un outil et la lecture d'une ressource. Une requête contient jsonrpc, id, method et éventuellement params. Une réponse contient jsonrpc, id et soit result, soit error. L'erreur comprend code, message et éventuellement data. Request messages = messages de requête ; Response messages = messages de réponse ; number = nombre ; string = chaîne de caractères ; object = objet ; unknown = type non déterminé. Le signe ? désigne un champ facultatif.

Définir le serveur de référence

Vous connaissez désormais l'architecture et le serveur qui servira de fil conducteur. Vous pouvez créer un serveur MCP minimal en Python et vérifier son initialisation en local. Il exposera deux outils, `add()` et `joke()`, une ressource et un modèle de prompt. Sa simplicité est volontaire : l'objectif est de disposer d'un serveur testable de bout en bout dès maintenant, que vous pourrez enrichir sans modifier son cœur.

Le client pourra demander un calcul rapide ou une blague sur Chuck Norris, et lire une petite ressource, par exemple un court fichier JSON de configuration. Le modèle de prompt fournira des instructions réutilisables pour obtenir une blague appropriée ou expliquer un calcul de manière cohérente. Nous suivrons quelques règles : valider les paramètres de `add()`, accepter un sujet textuel simple dans `joke()`, garder une ressource courte et statique, décrire clairement les outils pour faciliter leur sélection et éviter les effets de bord cachés.

2

Préparer et créer un serveur MCP minimal en Python

Vous connaissez les rôles de l'hôte, du client et du serveur, ainsi que la séquence initialize puis initialized. Il est temps de faire fonctionner un premier serveur. Procédez par étapes courtes : créez la structure du projet, ajoutez un ou deux outils et vérifiez la connexion. Le gestionnaire de paquets uv accélère les installations et fournit un moyen simple d'exécuter le projet, sans changer votre manière d'écrire du Python. Il réduit surtout les attentes et les manipulations liées aux environnements virtuels.

Procédez ainsi :

— Créez un dossier pour le serveur avec uv init .
— Installez le SDK, ses outils en ligne de commande et requests : uv add "mcp[cli]" requests.
— Gardez une structure simple : un fichier server.py à la racine du projet suffit pour commencer.
— Renseignez le fichier .gitignore et décrivez les outils dans le README pour conserver les informations utiles.

Pourquoi uv ? À ce stade, vous avez surtout besoin de retours rapides. Des installations qui prennent quelques secondes permettent de recréer facilement un environnement propre en cas de problème. Vous relancerez souvent le serveur pour ajuster les paramètres et les descriptions des outils. Une boucle de développement courte est alors plus utile qu'une architecture sophistiquée.

Vous utiliserez également MCP Inspector pour vérifier le serveur dans une interface graphique. Inspector étant écrit en TypeScript, Node.js doit être installé sur votre machine. Cela ne change rien au choix de Python pour le serveur : Node.js sert ici uniquement à exécuter un outil de développement.

Dans notre exemple, nous conserverons tout dans un seul dépôt. Nous créerons le projet avec uv init et installerons mcp[cli] pour lancer et déboguer le serveur avec une seule commande. Nous réorganiserons le code lorsque le nombre d'outils le justifiera. Voici les commandes de départ :

```
uv init <project-name>
uv add "mcp[cli]" requests
```

Si vous préférez ne pas utiliser uv, suivez la même démarche avec un environnement virtuel, puis exécutez le serveur avec Python :

```
python -m venv venv
source ./venv/bin/activate
pip install "mcp[cli]" requests
```

Vérifiez ensuite le serveur avec Inspector. Nous détaillerons cette étape un peu plus loin.

Créer le fichier server.py

Une fois le projet prêt, créez server.py à sa racine si ce n'est pas déjà fait. Avant de passer à l'exemple de code, voyons ce que ce fichier doit contenir pour que vous puissiez l'adapter à votre environnement. Il rassemblera les outils, les ressources et les prompts, importera les composants du SDK Python, créera l'instance du serveur et définira son cycle de vie.

Gardez ce fichier court : déclarez le serveur, enregistrez ses outils, ses ressources et ses prompts, puis raccordez les fonctions liées à son cycle de vie. Placez la logique métier dans des fonctions que vous pourrez tester séparément. Une identité et des capacités clairement définies faciliteront la découverte du serveur par un client utilisant un LLM.

Accordez une attention particulière au cycle de vie. Une initialisation incomplète peut empêcher le client de lister les outils ou produire des appels sans effet apparent. Le SDK prend en charge JSON-RPC : vous n'avez pas à implémenter les échanges de bas niveau. En revanche, consignez dans les journaux la réception de initialize et le moment où le serveur est prêt. Vous pourrez ainsi vérifier rapidement le démarrage du processus et le transport avant de rechercher un problème dans le code des outils.

Implémenter les premiers outils

Commencez par add et joke. Des outils courts et explicites seront plus faciles à sélectionner pour le LLM. Définissez des paramètres clairs et une description compréhensible. Cette description joue un rôle fonctionnel : elle permet au modèle de relier la demande de l'utilisateur à la bonne fonction.

— add() reçoit deux nombres et renvoie leur somme. Utilisez deux paramètres entiers clairement nommés, a et b. Exemple de description : « Additionne deux nombres et renvoie a + b. »
— joke() interroge une API externe et renvoie une blague sur une ligne, éventuellement filtrée selon un sujet. Vous pouvez accepter quelques catégories appropriées ou un sujet en texte libre. Exemple de description : « Récupère une courte blague sur Chuck Norris. Un sujet facultatif permet de préciser la recherche. »

L'outil joke repose volontairement sur une API : il représente ainsi une intégration réelle dont les résultats changent d'un appel à l'autre. Les critères de réussite sont donc les suivants :

— L'appel aboutit.
— La réponse contient une ligne de texte non vide.
— En cas d'échec, un message clair explique le problème à l'utilisateur : délai dépassé, limite de requêtes atteinte, etc.

Ne testez pas le texte exact de la blague. Vérifiez le format de la réponse et une qualité minimale.

Pour rédiger des descriptions utiles aux LLM, commencez par un verbe précis, indiquez l'effet principal de l'outil ou le format de son résultat, puis expliquez les paramètres essentiels : unités, format attendu, valeurs autorisées. Privilégiez des termes simples. Une description directe facilite la sélection de l'outil.

Dans le serveur de référence, implémentez add(a : int, b : int) et joke(topic : str | None), qui interrogera l'API publique de blagues sur Chuck Norris. Vous pouvez partir de cet exemple :

```python
import random
import requests
from mcp.server.fastmcp import FastMCP

mcp = FastMCP("First server")

@mcp.tool()
def add(a: int, b: int) -> int:
    return a + b

@mcp.tool()
def joke(topic: str | None = None) -> str:
    if topic:
        res = requests.get(
            "https://api.chucknorris.io/jokes/search",
            params={"query": topic},
        )
        data = res.json()
        results = data.get("result", [])
        if results:
```

```
            return random.choice(results).get("value", "No joke found.")
    res = requests.get("https://api.chucknorris.io/jokes/random")
    data = res.json()
    return data.get("value", "No joke found.")

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

Vous pourrez ensuite valider les entrées, intercepter les erreurs réseau et renvoyer un message compréhensible plutôt que laisser remonter une exception. Pour l'instant, limitez simplement le temps d'attente des requêtes ; vous approfondirez les questions de latence plus tard.

Ajouter une ressource et un prompt

Exposez une ressource statique et un modèle de prompt afin que le client découvre autre chose que des outils. Les ressources apportent du contexte à lire : fichiers, paramètres ou schémas. Les prompts fournissent des instructions réutilisables pour orienter le modèle.

— Enregistrez une ressource statique, par exemple resources/readme.txt, qui explique brièvement le rôle du serveur et donne quelques exemples d'appels. Rendez-la accessible à la découverte et à la lecture par le client. Vous pourrez ainsi fournir des instructions de référence sans les recopier dans chaque conversation. Exemple de contenu :

Serveur MCP de référence

Outils disponibles :

add(a : int, b : int) -> int

joke(topic : str) -> str

Utilisez ce serveur pour tester la sélection des outils, le passage des arguments et les échanges MCP de base.

— Créez un modèle de prompt, par exemple math_add_prompt, avec une consigne structurée : « Lorsque l'utilisateur demande une addition, appelle l'outil add avec des entiers. Confirme les valeurs avant l'exécution. » Restez bref et précis, comme dans cet exemple :

Tu utilises le serveur MCP de référence.

Lorsque l'utilisateur demande une addition, appelle add avec les entiers a et b. Confirme les valeurs avant l'exécution. Réponds uniquement par le résultat numérique.

Lorsque l'utilisateur demande une blague, appelle joke avec le sujet indiqué, ou sans sujet s'il n'en précise aucun. Réponds uniquement par le texte de la blague.

Rendez la ressource et le prompt accessibles au client. Un client intégrant un LLM pourra les découvrir et les ajouter au contexte pour choisir les outils plus régulièrement. Des instructions claires, accompagnées du contexte utile, réduisent les erreurs. Voici le code de l'exemple complété :

```python
import random
import requests
from mcp.server.fastmcp import FastMCP

mcp = FastMCP("First server")

@mcp.tool()
def add(a: int, b: int) -> int:
    """Add two integers."""
    return a + b

@mcp.tool()
def joke(topic: str | None = None) -> str:
    """Return a Chuck Norris joke; optionally filter by topic."""
    if topic:
        res = requests.get(
            "https://api.chucknorris.io/jokes/search",
            params={"query": topic},
        )
        data = res.json()
        results = data.get("result", [])
        if results:
            return random.choice(results).get("value", "No joke found.")
    res = requests.get("https://api.chucknorris.io/jokes/random")
    data = res.json()
    return data.get("value", "No joke found.")

@mcp.resource("reference:/readme")
def readme() -> str:
    """How to use this server and sample calls."""
    return (
        "Reference MCP Server\n\n"
        "Tools:\n"
        "- add(a: int, b: int) -> int\n"
        "- joke(topic: str) -> str\n\n"
        "Use this server to test tool selection, argument passing, "
        "and basic MCP wiring.\n"
    )

@mcp.prompt(
    name="math_add_prompt",
    description="Recipe for using add/joke reliably with this server.",
)
def math_add_prompt() -> str:
    return (
        "You are using the Reference MCP server.\n\n"
        "When the user asks to add numbers:\n"
        "1) Call the add tool with integers a and b.\n"
        "2) Confirm inputs before executing.\n"
        "3) Reply with just the numeric result.\n\n"
        "If the user asks for a joke:\n"
        "1) If they provide a topic, call joke with that topic. "
```

```
        "Otherwise call joke with no topic.\n"
        "2) Reply with the joke text only.\n"
    )

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

Exécuter le serveur et le vérifier dans MCP Inspector

Commencez par exécuter le serveur avec stdio. C'est le chemin le plus rapide, avec peu de paramètres susceptibles d'être mal configurés. Utilisez les commandes du SDK pour éviter d'écrire vous-même le code du transport.

— Avec uv : uv run mcp dev server.py
— Avec pip et un environnement virtuel : mcp dev server.py

Cette commande lance Inspector et votre serveur en stdio. Vérifiez que l'initialisation se termine et que le serveur devient disponible. Si la liste des outils ne s'affiche pas, contrôlez le chemin du fichier, le répertoire de travail et l'accès du processus à l'entrée et à la sortie standard. À ce stade, le serveur fonctionne en local avec stdio, pas avec HTTP.

Voici les premiers points à vérifier :

— La liste des outils contient-elle add et joke ?
— Un appel à add avec de petits entiers renvoie-t-il la somme attendue ?
— joke renvoie-t-il une ligne de texte et, en cas d'échec, un message clair ?
— Pouvez-vous découvrir et lire la ressource statique, puis consulter le modèle de prompt ?

Corrigez les problèmes détectés avant d'ajouter d'autres fonctionnalités. Cette première boucle doit être fiable.

Inspector permet de se connecter au serveur stdio, de lister ses capacités et d'appeler ses outils. Vous voyez ainsi ce que recevra le client et pouvez repérer des incohérences dans les paramètres ou les descriptions.

Pour maîtriser plus précisément le lancement d'Inspector, exécutez-le directement et laissez-le démarrer le serveur comme sous-processus :

```
npx @modelcontextprotocol/inspector
```

Dans les paramètres de connexion d'Inspector, indiquez ensuite la commande de lancement du serveur, par exemple uv run mcp dev server.py ou python server.py.

Quel que soit le mode de lancement retenu, lorsque l'interface s'affiche dans le navigateur, cliquez sur Connect pour démarrer le serveur et terminer l'initialisation MCP.

Remarque sur la connexion à Inspector

Inspector fonctionne par l'intermédiaire d'un proxy et peut demander une clé de sécurité. Si Connect ne fonctionne pas, recherchez cette clé dans la sortie du terminal, ouvrez Configuration dans Inspector et collez-la dans le champ prévu. Relancez ensuite la connexion.

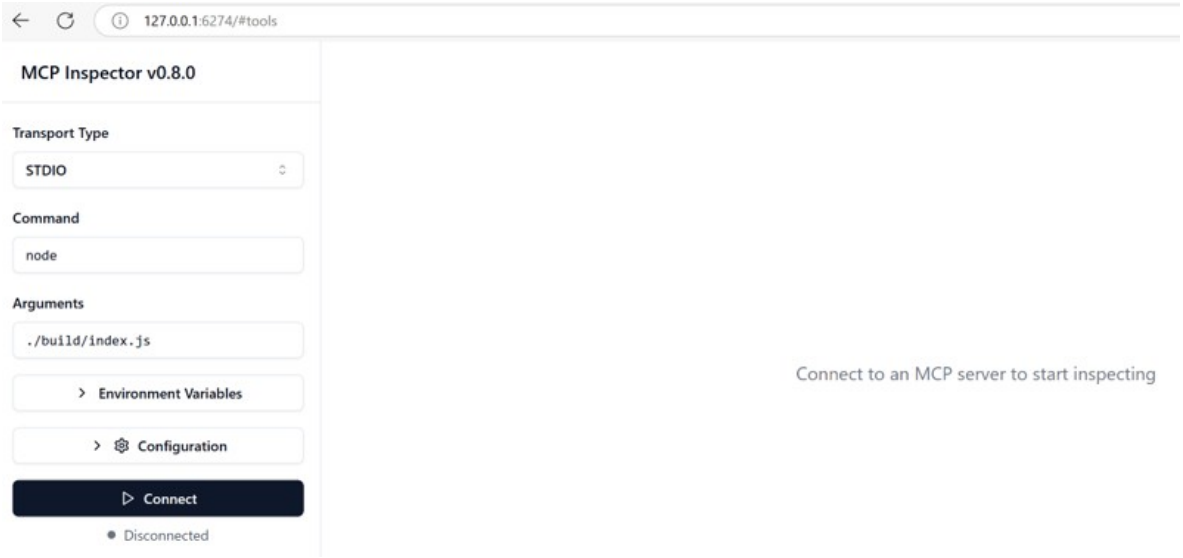

FIGURE 2.1 : L'interface de MCP Inspector.

— Ouvrez le panneau Tools et demandez la liste des outils. Vous devez retrouver add et joke avec leurs descriptions. Si un nom est inadapté ou une description trop vague, corrigez le code et reconnectez-vous.
— Appelez add et joke depuis l'interface. Essayez add avec 18 et 2, puis cliquez sur Run Tool. Testez ensuite joke avec un sujet comme « cats ». Vérifiez que les résultats figurent dans la réponse de l'outil, et pas uniquement dans les journaux.
— Ouvrez les panneaux Resources et Prompts. Lisez la ressource statique et le modèle de prompt pour confirmer qu'ils sont accessibles.

Inspector peut aussi fonctionner en ligne de commande pour les contrôles de CI. Indiquez le mode sans interface et la méthode à exécuter, par exemple lister les outils ou appeler un outil, puis analysez le résultat JSON dans votre pipeline. Voici quelques exemples :

— Lister les outils du serveur local en stdio :

```
npx @modelcontextprotocol/inspector-cli uv run mcp run server.py --method tools/list
```

— Appeler add, dont le résultat est déterministe :

```
npx @modelcontextprotocol/inspector-cli uv run mcp run server.py --method tools/call --tool-name add --tool-arg a=18 --
```

— Appeler joke, qui dépend d'une API :

```
npx @modelcontextprotocol/inspector-cli uv run mcp run server.py --method tools/call --tool-name joke --tool-arg topic=
```

Inspector écrit sur la sortie standard un résultat JSON exploitable par un programme. Il convient donc aux vérifications locales rapides et aux contrôles de CI. Ces tests permettent notamment de repérer une régression après un changement de nom d'outil ou de type de paramètre.

Vérifications dans l'interface graphique

La connexion s'établit sans erreur et la liste des outils s'affiche.

Les descriptions expriment clairement ce que fait chaque outil.

Les appels réussissent avec des entrées simples et réalistes.

La ressource et le prompt sont visibles et lisibles.

Votre serveur fonctionne et a été vérifié dans Inspector. Vous pouvez maintenant améliorer la fiabilité de la sélection en affinant les noms d'outils, les schémas, les ressources et les modèles de prompt.

3

Concevoir des outils des ressources et des prompts que l'IA choisit correctement

---

La structure du serveur est en place et l'initialisation fonctionne. Il faut maintenant aider l'IA à choisir le bon outil dès la première tentative. Traitez les noms, les descriptions et les paramètres comme une API publique destinée au LLM. Une formulation précise peut faire toute la différence entre un appel correct et un choix approximatif.

— Noms et descriptions : choisissez des noms courts et concrets, puis utilisez la description pour préciser l'intention. add et joke conviennent très bien si leur description résume clairement la capacité et permet de la rapprocher d'une demande utilisateur.
— Verbes ambigus : évitez handle, process, run ou do_it. Préférez des verbes liés au domaine, comme add, fetch, book, list ou save. Si deux outils semblent répondre à la même demande, cette ambiguïté risque de se retrouver dans les choix du modèle.

— Correspondance entre intention et paramètres : utilisez des noms proches du vocabulaire de l'utilisateur. Si la demande porte sur un sujet, le paramètre topic est cohérent. Ajoutez une courte description pour indiquer au LLM quand le renseigner.

Pour le serveur de référence, retenez des noms explicites, rédigez une description courte pour chaque outil et adaptez les paramètres aux demandes attendues. Par exemple : « Additionne deux entiers et renvoie leur somme » et « Récupère une blague aléatoire sur Chuck Norris, éventuellement filtrée par sujet ». Utilisez a et b pour les nombres, topic pour le sujet des blagues. Le nom seul ne suffit pas toujours ; une description claire améliore nettement la correspondance avec l'intention.

Comme nous l'avons vu, l'appel d'outils étend le principe de l'appel de fonctions. Il standardise des capacités pour qu'un client MCP intégrant un LLM puisse choisir la bonne fonction de manière cohérente. Des descriptions vagues entraînent aussi bien des appels inutiles que des outils ignorés à tort. Affinez les formulations jusqu'à ce qu'un collègue puisse associer immédiatement la liste des outils aux demandes courantes. Le modèle s'appuie lui aussi sur ces indices.

Clarifier les paramètres et les schémas

Des schémas solides facilitent la sélection et sécurisent l'exécution. Précisez les types et les champs obligatoires lorsque le modèle ne doit pas deviner, et définissez un format de résultat fiable pour le client. Limitez le nombre de paramètres : chaque champ supplémentaire crée une nouvelle possibilité d'erreur.

— Types et champs obligatoires : utilisez des types explicites et ne rendez obligatoires que les données indispensables. Pour un paramètre facultatif, indiquez-le et décrivez le comportement par défaut.
— Validez les entrées dès le départ et expliquez les erreurs simplement. Un message comme « Le paramètre a doit être un entier » pourra aider le LLM à corriger son prochain appel. Évitez les traces d'exécution et le jargon interne dans la réponse destinée à l'utilisateur.
— Format des résultats : gardez une structure stable. Si l'outil renvoie un objet, conservez les mêmes clés et les mêmes types pour que les clients puissent afficher les résultats ou les réutiliser dans un traitement.

Liste de vérification des schémas

Nommez et décrivez les paramètres avec le vocabulaire de l'utilisateur : topic plutôt que categoryId, sauf si le domaine utilise réellement des identifiants.

Définissez les types et le nombre minimal de champs obligatoires. Rendez les autres facultatifs.

Précisez brièvement quand utiliser chaque paramètre.

Définissez une structure claire pour les réussites et pour les erreurs.

Documentez les valeurs par défaut et les cas limites : chaîne vide, valeur nulle ou sujet inconnu.

Vous pourriez renforcer l'exemple en validant les entrées et en renvoyant des objets JSON structurés, sous forme de dictionnaires Python. add imposerait alors deux entiers, a et b, et renverrait toujours { “sum” : }. Avec un paramètre topic facultatif, joke renverrait { “joke” : , “topic” : <string|null> }.

En cas d'entrée invalide, fournissez un message compréhensible, par exemple : « Sujet indisponible ; essayez travel ou laissez le champ vide. » Pensez autant aux échanges entre le modèle et le serveur qu'à la lecture humaine.

Utiliser les ressources pour fournir du contexte

Les ressources rassemblent le contexte statique — paramètres, schémas, documents — que le client peut lire et transmettre au modèle. Elles permettent de raccourcir les prompts tout en étayant les réponses. Si une ressource en lecture seule suffit, il n'est pas nécessaire de tout faire passer par un outil.

— Contenu des ressources : préférences utilisateur, règles internes, schémas, exemples et documentation de référence. Tout ce que l'IA doit lire sans le modifier peut y figurer. En local, un serveur MCP de fichiers peut exposer des dossiers ou des fichiers ; à distance, vous pouvez fournir des documents structurés par HTTP.
— Références dans les prompts : gardez les instructions courtes et mentionnez les ressources par leur nom ou leur chemin. Un modèle peut, par exemple, indiquer : « Utilise la ressource pricing_policy pour calculer les totaux. »
— Gestion des versions : indiquez une version dans le nom ou le chemin de la ressource, ou dans un champ de son contenu. Gardez les versions stables pour que les appels répétés et les données mises en cache restent cohérents.

Sans ressources, le contexte se fragmente souvent entre de nombreux prompts dans lesquels on recopie les mêmes informations. Centralisez les données statiques et laissez le client les récupérer au besoin. Cela rejoint l'image de MCP comme un connecteur universel pour les applications à agents : les outils effectuent les opérations, les ressources informent et les prompts organisent leur utilisation. N'intégrez pas de secrets aux ressources et protégez les accès sensibles avec les mécanismes de transport et d'authentification présentés plus loin.

Concevoir des modèles de prompt réutilisables

Un modèle de prompt fournit une méthode réutilisable qui aide l'utilisateur et le LLM à obtenir le résultat attendu avec moins de tokens. Gardez-le court, prévoyez des emplacements pour les variables importantes et expliquez quand utiliser un outil sans imposer trop de contraintes au modèle.

— Instructions et variables : deux ou trois phrases et une courte liste de variables suffisent souvent. Exemple : « Pour récupérer une blague aléatoire, tu peux préciser un sujet. Utilise de préférence les mots employés par l'utilisateur. » Variable : topic.
— Concision : tenez compte de la fenêtre de contexte et évitez les longues consignes répétitives. Si des explications supplémentaires sont nécessaires, placez-les dans une ressource, par exemple reference :/readme, que le client chargera pour enrichir le contexte.
— Orientation vers un outil : une courte indication peut être utile lorsque plusieurs outils ont des descriptions proches. « Utilise joke lorsque l'utilisateur demande une blague » peut améliorer la précision sans figer tout le comportement.

La qualité du choix se prépare avant l'exécution. Des instructions brèves et des variables bien définies aident le modèle à traduire la demande en paramètres, puis à sélectionner le bon outil. Des modèles trop contraignants peuvent produire l'effet inverse. Laissez les descriptions et les schémas porter l'essentiel des précisions.

Tester avec MCP Inspector

Utilisez Inspector dans le navigateur et en ligne de commande pour accélérer les essais et automatiser les contrôles de CI. Vous pourrez examiner les schémas, tester les cas courants et explorer les cas limites sans créer une interface complète.

— Lister et examiner les schémas : lancez le serveur avec Inspector, connectez-vous, puis affichez les outils. Vérifiez leurs noms, leurs descriptions et leurs paramètres. Les formulations ambiguës deviennent alors plus visibles.

Après toute modification d'un nom, d'une description ou d'un schéma, relancez immédiatement la commande suivante :

```
npx @modelcontextprotocol/inspector-cli uv run mcp run server.py --method tools/list
```

Si la liste ne reflète pas vos modifications, vous testez probablement un autre point d'entrée, un processus qui n'a pas été relancé ou une connexion qui n'a pas été rétablie.

— Tester les cas limites : essayez les paramètres facultatifs, l'absence de champs obligatoires et les valeurs hors du domaine attendu. Vous devez obtenir des erreurs lisibles et des résultats de structure stable. En ligne de commande, indiquez la méthode à exécuter et récupérez directement la sortie pour vos scripts de CI. Voici deux appels de base à joke que vous pouvez adapter :
— Appeler joke sans sujet :

```
npx @modelcontextprotocol/inspector-cli uv run mcp run server.py --method tools/call --tool-name joke
```

— Appeler joke avec un sujet :

```
npx @modelcontextprotocol/inspector-cli uv run mcp run server.py --method tools/call --tool-name joke --tool-arg topic
```

Ces appels doivent renvoyer une réponse correctement structurée, par exemple :

```
{
  "content": [
    {"type": "text", "text": "...texte de la blague..."}
  ],
  "structuredContent": {"result": "...texte de la blague..."},
  "isError": false
}
```

— Lire les ressources et les modèles : vérifiez que les ressources s'affichent correctement et que les prompts les désignent par leur nom. Utilisez des libellés courts pour faciliter leur repérage dans le client.

Pour notre exemple, suivez cette séquence :

— Ouvrez Inspector, connectez-vous au serveur Python en stdio et affichez la liste des outils.
— Appelez add en renseignant a et b.
— Passez une chaîne de caractères à a pour vérifier la validation.
— Appelez joke avec puis sans topic.
— Passez en ligne de commande, exécutez tools/list et appelez joke pour vérifier que les contrôles scriptés fonctionnent.

Rappel : Inspector est écrit en TypeScript. Node.js doit être disponible, même si votre serveur est écrit en Python.

Appliquer ces principes à d'autres domaines

Les mêmes principes s'appliquent à une démonstration ludique et à une application métier. Prenons une application de réservation de voyage : vols, hôtels, services bancaires et fichiers locaux peuvent relever de serveurs distincts, chacun proposant des outils, des ressources et des prompts clairement définis. Vous composez ainsi plusieurs services.

— Si joke est correctement sélectionné, vous avez validé votre manière de nommer les outils, de les décrire et de définir leurs schémas. Appliquez la même méthode à book_flight, reserve_hotel et pay_invoice. Chaque nom annonce une capacité ; chaque description précise le contrat présenté au LLM.
— Certaines capacités, comme la lecture et l'écriture de fichiers locaux en stdio, doivent rester locales. D'autres peuvent être distantes. Pour ces dernières, utilisez Streamable HTTP et ne conservez l'ancien transport SSE que pour assurer une compatibilité temporaire.
— Séparez les serveurs selon les domaines et les permissions. Si les opérations bancaires exigent une authentification et un audit plus stricts que la recherche d'hôtels, placez-les sur un serveur distinct. Cette démarche ressemble à la décomposition d'un monolithe en services ; MCP simplifie leur raccordement.

Une fois la sélection fiabilisée, vous pouvez intégrer le serveur à des applications hôtes avec mcp.json et tester une conversation complète.

4

Intégrer le serveur à VS Code et à Claude Desktop

Votre serveur MCP Python est prêt. Il faut maintenant le rendre accessible aux applications hôtes. Un petit fichier de configuration, mcp.json, permet à l'hôte de découvrir le serveur et ses capacités. Comme lorsqu'on branche un périphérique, l'hôte peut établir la connexion MCP et découvrir les outils, les ressources et les prompts. Une configuration simple et explicite facilite cette intégration.

Pour limiter la configuration au projet, placez mcp.json dans un dossier .vscode à la racine. Vous pourrez le versionner avec le code et réduire les écarts entre environnements. Certains hôtes acceptent aussi une configuration utilisateur globale, mais commencez à l'échelle du projet pour tester rapidement les changements sans affecter les autres espaces de travail.

Ajoutez un objet servers contenant une entrée pour chaque serveur que l'hôte doit lancer. Pour un serveur local en stdio, renseignez command et args. command désigne le programme de lancement, par exemple uv ou python. args doit contenir le chemin exact du point d'entrée du serveur. Exemples :

— Si server.py se trouve à la racine du projet :

```
"args": ["run", "server.py"]
```

— Si server.py se trouve dans src/ :

```
"args": ["run", "src/server.py"]
```

L'hôte utilise ces paramètres pour démarrer le processus et raccorder automatiquement l'entrée et la sortie standard. Il effectue ensuite l'échange d'initialisation.

Dans l'exemple local en stdio présenté ici, le type n'est pas précisé. La configuration peut prendre cette forme :

```
{
  "inputs": [],
  "servers": {
    "demo-server": {
      "command": "uv",
      "args": ["run", "mcp", "run", "server.py"]
    }
  }
}
```

Pour un serveur SSE et un serveur local sous Windows, voici un exemple de configuration :

```
{
  "inputs": [],
  "servers": {
    "sse-server": {
      "type": "sse",
      "url": "http://localhost:4321/sse"
    },
    "mcp-demo": {
      "command": "cmd",
      "args": ["/c", "node", "\\path\\to\\server.js"]
    }
  }
}
```

La section inputs permet de déclarer des valeurs que l'hôte doit demander de manière sécurisée à la connexion, comme une clé d'API ou un jeton bearer si le serveur exige une authentification. Gardez des noms stables et décrivez clairement l'usage de chaque valeur. Considérez mcp.json comme un contrat de configuration entre l'hôte et le serveur : utilisez des chemins explicites, évitez les détours inutiles et documentez le rôle de chaque entrée.

Utiliser le mode Agent de VS Code

Si vous utilisez VS Code avec GitHub Copilot, enregistrez le serveur dans mcp.json pour que l'agent puisse découvrir et appeler ses outils. Si vous utilisez Claude Desktop, passez à la section suivante.

Une fois mcp.json en place, utilisez la conversation intégrée de GitHub Copilot en mode Agent pour transformer vos demandes en appels d'outils. L'hôte découvre les capacités, sélectionne les outils et demande l'autorisation avant leur exécution. De votre côté, fournissez un chemin de lancement correct et des descriptions d'outils précises.

Démarrer et connecter le serveur

Ouvrez le projet dans VS Code. Vérifiez que .vscode/mcp.json contient le serveur dans servers, avec les bonnes valeurs pour command et args. Dans l'éditeur du fichier mcp.json, démarrez le serveur à l'aide de la commande intégrée affichée au-dessus de son entrée.

Figure 4.1 : Démarrage du serveur depuis la commande intégrée à VS Code.

L'hôte lance le processus en stdio, termine l'initialisation et affiche l'état connecté. Ouvrez la vue Chat, puis choisissez Agent dans le menu du mode de conversation pour permettre au LLM d'utiliser les outils MCP.

Activer ou désactiver des outils

L'option Configure Tools affiche les outils découverts sur le serveur, avec leurs noms et leurs descriptions. Vous pouvez activer ou désactiver chaque outil pour le projet. Cela permet d'isoler un outil pendant le débogage ou de ne rendre disponible qu'une partie des fonctionnalités lors d'un déploiement progressif.

Autoriser l'exécution d'un outil

Dans la conversation, formulez une demande simple, par exemple « additionne 18 et 2 ». Le LLM extrait les arguments en s'appuyant sur le schéma, puis l'hôte propose un appel d'outil et vous demande de l'autoriser. Cliquez sur Allow pour continuer. Si les arguments sont mal interprétés, reformulez ou utilisez des chiffres : le modèle s'appuie principalement sur le schéma et la description.

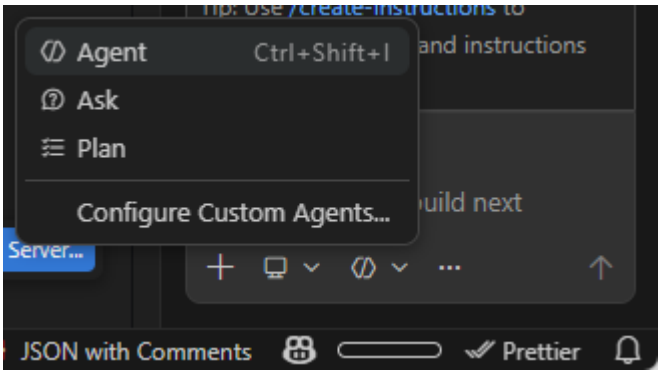

Figure 4.2 : Sélection du mode Agent dans le panneau Chat de VS Code.

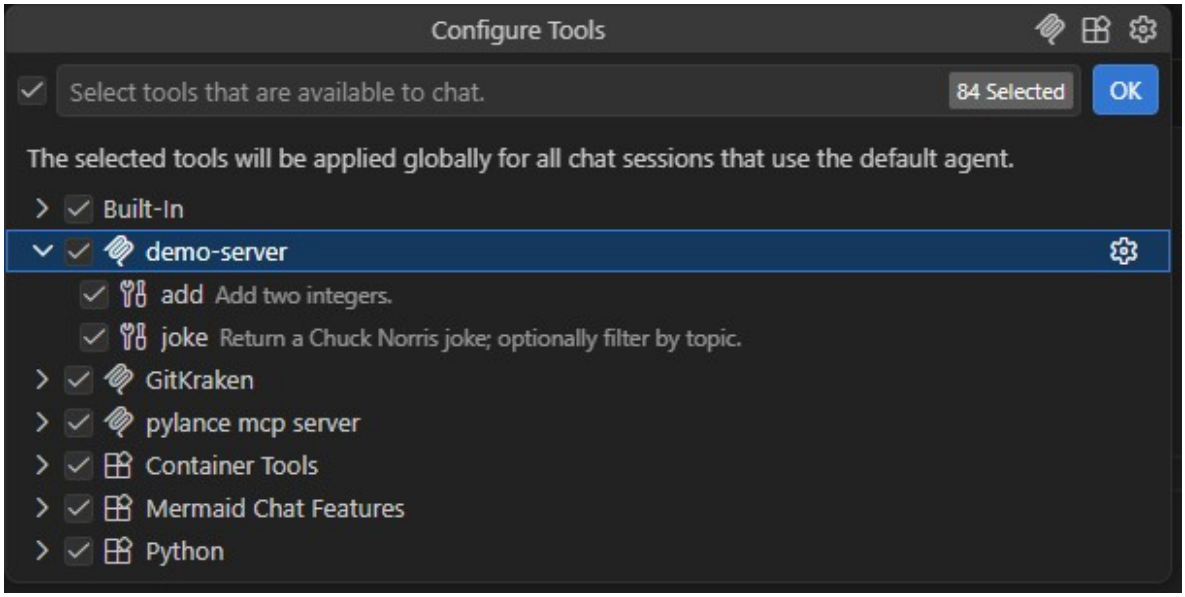

Figure 4.3 : Activation et désactivation des outils avec Configure Tools dans VS Code.

Configurer Claude Desktop

Claude Desktop est une application hôte MCP. Il peut démarrer les serveurs configurés, découvrir leurs outils et leurs ressources, puis demander l'autorisation avant les appels. Avec une configuration claire et un chemin de lancement stdio exact, le comportement doit être proche de celui observé dans VS Code.

Installez Claude Desktop, puis ajoutez dans claude_desktop_config.json la définition du serveur utilisée pour VS Code, sous la clé mcpServers. Exemple :

```
{
  "mcpServers": {
    "demo-server": {
      "type": "stdio",
      "command": "uv",
      "args": ["run", "mcp", "run", "server.py"]
    }
  }
}
```

Redémarrez Claude Desktop pour recharger la configuration. L'hôte découvre les serveurs, les lance à partir de command et args, puis termine l'initialisation. Vous pouvez alors vérifier la connexion et tester les outils depuis la conversation.

Les libellés de l'interface peuvent varier légèrement. Une même phrase peut aussi conduire à une interprétation différente selon l'hôte. Si un hôte extrait mal les nombres écrits en lettres, utilisez des chiffres ou précisez les descriptions des paramètres côté serveur.

Tester la chaîne complète

Une fois VS Code ou Claude Desktop configuré, vérifiez tout le parcours : découverte, sélection, exécution et restitution du résultat. Commencez avec des demandes simples pour add et joke. Écrivez « additionne 18 et 2 » ou « calcule 18 + 2 ». L'hôte doit proposer add avec les arguments correspondants et demander votre autorisation. Acceptez et vérifiez le résultat. Demandez ensuite « une blague sur Chuck Norris et les chats ». L'hôte doit sélectionner joke, renseigner topic et afficher la réponse. S'il choisit mal, améliorez la description jusqu'à obtenir une sélection régulière du bon outil.

Facilitez le travail de l'hôte avec des noms de paramètres clairs, des types précis et des descriptions courtes. Si « deux » est mal compris, écrivez 2. Si le sujet est ambigu, ajoutez des exemples de valeurs dans la description. Demandez aussi la lecture d'une ressource au cours de la même conversation. Son contenu doit rester court et pertinent : vous vérifiez la récupération de contexte à la demande, sans saturer la fenêtre du modèle.

Conseils d'exploitation

Utilisez mcp.json comme point de configuration des serveurs du projet. Une configuration claire facilite la gestion de plusieurs serveurs, limite la portée des secrets et évite les mauvaises surprises lorsque des collègues clonent le dépôt. Ce chapitre utilise principalement stdio. Le passage à Streamable HTTP

modifiera le mode de connexion, mais conservera les mêmes mécanismes de découverte des outils, ressources et prompts. Préférez un fichier .vscode/mcp.json par projet, avec une entrée explicite sous servers pour chaque serveur. Si plusieurs serveurs proposent des outils de noms proches, rendez leurs descriptions et leurs paramètres suffisamment distincts pour éviter les erreurs de sélection.

Les premiers essais peuvent échouer. Les causes fréquentes sont un mauvais chemin dans args, un programme de lancement incorrect dans command ou une description trop vague. Vérifiez d'abord le lancement : sans lui, l'échange initialize/initialized ne peut pas avoir lieu. Affinez ensuite les descriptions et les schémas. Des métadonnées précises résolvent une grande partie des erreurs de sélection.

À mettre en pratique

— Créez .vscode/mcp.json avec une entrée stdio dans servers, le programme uv ou python et le chemin exact vers server.py.
— Connectez le serveur dans VS Code, activez uniquement add et joke, puis testez les deux dans la conversation en mode Agent en autorisant leur exécution.
— Reproduisez les tests dans Claude Desktop. En cas d'erreur d'interprétation des arguments, précisez les descriptions des outils.
— Demandez la lecture de la ressource dans la conversation pour vérifier la récupération du contexte.
— Préparez la migration en recensant les serveurs que vous exposerez plus tard avec Streamable HTTP, sans modifier encore le code.

Lorsque les essais locaux sont fiables, créez votre propre client pour maîtriser les prompts, les autorisations et l'expérience utilisateur.

5

Créer un client qui s'appuie sur un LLM

Vous avez créé un serveur MCP en Python et l'avez vérifié avec Inspector et une application hôte. Vous pouvez maintenant concevoir votre propre client pour offrir l'expérience adaptée à votre produit. Le protocole restera le même, mais vous maîtriserez le transport, les prompts et l'observabilité. L'image de MCP comme connecteur universel prend ici tout son sens : une fois l'interface comprise, vous pouvez l'intégrer à différents environnements.

Les hôtes conviennent aux développeurs, mais vos utilisateurs n'y travailleront pas nécessairement. Un client dédié permet d'intégrer les actions MCP à votre interface, d'adapter les textes et les protections à votre produit et d'éviter les changements d'outil. Vous décidez aussi quand orienter le modèle ou exiger une autorisation explicite. Certaines tâches passent mieux par des événements que par une conversation : le client peut fonctionner sans interface, écouter une file de messages et choisir un outil avec ses arguments pour chaque événement. Cette approche convient notamment aux environnements cloud, où le client s'exécute dans une application web ou un processus de traitement, et les serveurs en local ou à distance.

Vous êtes responsable de l'expérience proposée. MCP définit l'initialisation et les schémas d'outils. Vous décidez quel outil appeler, quand l'appeler, comment présenter son résultat et comment réagir aux erreurs du modèle.

Démarrer la connexion et négocier les capacités

Au démarrage, le client doit lancer ou localiser un serveur, puis terminer l'initialisation. Commencez simplement : en développement, démarrez le serveur Python comme sous-processus en stdio. En préproduction ou en production, connectez-vous en Streamable HTTP, sans créer ce sous-processus. Réservez l'ancien transport SSE aux cas de compatibilité qui l'exigent.

Pour stdio, utilisez la même commande que dans le terminal, avec le chemin du module serveur. Pour Streamable HTTP, fournissez l'URL. Isolez cette logique dans une petite interface de transport afin de pouvoir changer de transport sans modifier le routage des appels ni le code du LLM.

Une fois le serveur démarré ou localisé, effectuez l'initialisation JSON-RPC. Le client envoie initialize avec ses fonctionnalités ; le serveur répond avec ses capacités ; le client envoie la notification initialized. Vous pouvez alors lister les outils, les ressources et les prompts, lire les ressources, récupérer les modèles et appeler les outils. Chaque session relie un client à un serveur, ce qui simplifie le suivi des échanges.

Voici un client stdio utilisant le SDK Python. Il démarre le serveur comme sous-processus, initialise la session, liste les outils et les ressources, lit une ressource, puis appelle un outil.

```
import asyncio
from mcp import ClientSession, StdioServerParameters
from mcp.client.stdio import stdio_client

# Démarrer le serveur en stdio avec uv.
server_params = StdioServerParameters(
    command="uv",
    args=["run", "mcp", "run", "server.py"],
    env=None,
)

async def run():
    async with stdio_client(server_params) as (read, write):
        async with ClientSession(read, write) as session:
            # Initialiser la connexion MCP.
            await session.initialize()

            # Lister les ressources disponibles.
            print("LISTING RESOURCES")
            resources = await session.list_resources()
            for resource in resources:
                print("Resource:", resource)

            # Lister les outils disponibles.
            print("LISTING TOOLS")
            tools = await session.list_tools()
```

```
            for tool in tools.tools:
                print("Tool:", tool.name)

            # Lire une ressource.
            print("READING RESOURCE")
            content, mime_type = await session.read_resource(
                "reference:/readme"
            )
            print("MIME TYPE:", mime_type)
            print(content)

            # Appeler un outil.
            print("CALL TOOL: add(a=1, b=7)")
            result = await session.call_tool(
                "add", arguments={"a": 1, "b": 7}
            )
            print("RESULT:", result.content)

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

Présenter les outils au LLM

Le client traduit les outils du serveur dans un format compréhensible par le LLM. Il commence par les lister, puis convertit leurs schémas dans le format de fonctions attendu par le modèle. Conservez les noms et les descriptions : ces dernières jouent souvent un rôle décisif dans la sélection.

Après l'initialisation, récupérez la liste des outils et mettez-la en cache jusqu'au redémarrage de la session. Avec plusieurs serveurs, maintenez un catalogue par serveur. Convertissez chaque outil en une définition de fonction reconnue par le LLM, en conservant son nom, sa description et son schéma de paramètres. Ne renommez pas les éléments exposés. L'exemple suivant transforme tools/list en définitions de fonctions, laisse le modèle choisir un outil et exécute tools/call avec ses arguments. Il utilise le SDK Azure Inference et la variable d'environnement GITHUB_TOKEN pour créer un AzureKeyCredential.

```
import asyncio
import json
import os
from mcp import ClientSession, StdioServerParameters, types
from mcp.client.stdio import stdio_client
from azure.ai.inference import ChatCompletionsClient
from azure.core.credentials import AzureKeyCredential

server_params = StdioServerParameters(
    command="uv",
    args=["run", "mcp", "run", "server.py"],
    env=None,  # Variables d'environnement facultatives.
)
```

```
def call_llm(prompt, functions):
    token = os.environ["GITHUB_TOKEN"]
    endpoint = "https://models.github.ai/inference"
    model_name = "openai/gpt-4o"
    client = ChatCompletionsClient(
        endpoint=endpoint,
        credential=AzureKeyCredential(token),
    )
    print("CALLING LLM")
    response = client.complete(
        messages=[
            {"role": "system", "content": "You are a helpful assistant."},
            {"role": "user", "content": prompt},
        ],
        model=model_name,
        tools=functions,
        # Paramètres facultatifs.
        temperature=1.0,
        max_tokens=1000,
        top_p=1.0,
    )
    response_message = response.choices[0].message
    functions_to_call = []
    if response_message.tool_calls:
        for tool_call in response_message.tool_calls:
            print("TOOL:", tool_call)
            name = tool_call.function.name
            args = json.loads(tool_call.function.arguments)
            functions_to_call.append({"name": name, "args": args})
    return functions_to_call

def convert_to_llm_tool(tool):
    tool_schema = {
        "type": "function",
        "function": {
            "name": tool.name,
            "description": tool.description,
            "parameters": {
                "type": "object",
                "properties": tool.inputSchema["properties"],
            },
        },
    }
    return tool_schema

async def run():
    async with stdio_client(server_params) as (read, write):
        async with ClientSession(read, write) as session:
            await session.initialize()
            resources = await session.list_resources()
            print("LISTING RESOURCES")
            for resource in resources:
```

```
                print("Resource:", resource)

            tools = await session.list_tools()
            print("LISTING TOOLS")
            functions = []
            for tool in tools.tools:
                print("Tool:", tool.name)
                print("Tool", tool.inputSchema["properties"])
                functions.append(convert_to_llm_tool(tool))

            prompt = "Add 2 to 20"
            # Demander au LLM quels outils appeler, le cas échéant.
            functions_to_call = call_llm(prompt, functions)
            for f in functions_to_call:
                result = await session.call_tool(
                    f["name"], arguments=f["args"]
                )
                print("TOOLS result:", result.content)

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

Acheminer les demandes vers les outils

La boucle doit rester claire : recueillir la demande, fournir au LLM la liste des fonctions converties, récupérer l'outil et les arguments proposés, exécuter l'appel sur le serveur MCP et restituer le résultat. Gardez le prompt court et concret ; les définitions de fonctions apportent déjà la structure nécessaire. Si votre application exige une autorisation avant l'exécution, interrompez la boucle pour la demander.

Validez les types à partir du schéma disponible. En cas d'échec, demandez au LLM de corriger les arguments selon ce même schéma ou affichez une erreur claire. Retrouvez l'outil d'origine par son nom et son serveur, puis appelez-le via le transport choisi. Dans notre exemple, « raconte-moi une blague sur les voyages » doit conduire à joke avec topic="travel". Validez, exécutez et restituez la blague. Voici une adaptation possible du code :

```
# Valider les arguments avant l'appel à un outil.
def validate_args(schema: dict, args: dict) -> tuple[bool, str]:
    required = schema.get("required", [])
    props = schema.get("properties", {})
    for k in required:
        if k not in args:
            return False, f"Missing required arg: {k}"
    for k, v in args.items():
        if (
            k in props
            and props[k].get("type") == "integer"
            and not isinstance(v, int)
        ):
            return False, f"Arg {k} must be integer"
    return True, ""
```

```python
async def route_prompt(
    prompt: str, functions: list[dict], session: ClientSession
):
    tool_calls = call_llm(prompt, functions)
    for call in tool_calls:
        fn = next(
            f for f in functions
            if f["function"]["name"] == call["name"]
        )
        schema = fn["function"]["parameters"]
        ok, err = validate_args(schema, call["args"])
        if not ok:
            # Demander une correction selon le même schéma.
            repair_prompt = (
                f"Fix tool args to match this JSON schema:\n{schema}"
                f"\nArgs:\n{call['args']}\nError: {err}"
            )
            repaired = call_llm(repair_prompt, functions=[fn])
            if repaired:
                call = repaired[0]

        # Insérer ici une demande d'autorisation si nécessaire.
        result = await session.call_tool(
            call["name"], arguments=call["args"]
        )
        return result.content
```

Traiter les réponses

Prévoyez les deux issues : résultat ou erreur. Transmettez les données de réussite sous une forme JSON que l'interface peut afficher. En cas d'échec, fournissez un état bref et une explication qui aide à agir. Affichez directement les résultats textuels ; pour les résultats structurés, proposez un court résumé et une option permettant de consulter le JSON brut.

Validez les arguments avant d'appeler les outils. Limitez les nouvelles tentatives aux défaillances temporaires et évitez de les multiplier. Présentez les erreurs clairement et conservez assez de contexte dans les journaux pour les reproduire, sans révéler de secrets : méthode, arguments expurgés des données sensibles, transport, résultat ou erreur, et durée. Commencez par ces éléments, puis complétez au besoin. Exemples :

Réussite avec texte : si joke(topic="travel") renvoie une chaîne, affichez-la dans la conversation. Vous pouvez ajouter une option « Afficher le JSON brut » pour consulter la réponse MCP complète.

Réussite avec données structurées : pour { "sum" : 20, "inputs" : {"a" : 18, "b" : 2} }, affichez un résumé tel que « Somme : 20 », puis une option d'accès au JSON brut.

Erreur : affichez un message court avec une piste de résolution, par exemple « Échec de l'API de blagues (502). Réessayez ou lancez l'appel sans sujet. » Dans les journaux, consignez les données nécessaires sans secrets : method=tools/call tool=joke args={topic :"travel"} transport=stdio status=error code=502 duration_ms=1830.

Combiner les environnements et les serveurs

Il est courant d'associer Python et Node.js : écrivez le serveur en Python et le client dans un environnement compatible MCP. Gardez l'adaptateur de transport léger pour conserver la même logique client entre stdio en local et Streamable HTTP en production. Si vous utilisez plusieurs serveurs, maintenez un registre indexé par identifiant de serveur, comprenant le transport et les outils découverts. Lorsqu'un outil est sélectionné, retrouvez son serveur et exécutez l'appel dans la session correspondante. Exemple :

```
servers = {
    "local": {
        "transport": "stdio",
        "start": ["uv", "run", "mcp", "run", "server.py"],
    },
    "prod": {
        "transport": "http",
        "url": "https://example.com/mcp",
    },
}

tool_index = {
    "local": ["add", "joke"],
    "prod": ["search", "summarize"],
}

def route_tool_call(tool_name, args, chosen_server_id, sessions):
    session = sessions[chosen_server_id]
    return session.call_tool(tool_name, arguments=args)
```

Si votre application manipule des données sensibles, des paiements ou du stockage local au moyen de plusieurs serveurs, privilégiez Streamable HTTP pour les services déployés à distance. Le chapitre suivant aborde cette migration et la sécurité : authentifier l'accès aux serveurs exposés sur Internet et demander une autorisation avant les outils qui écrivent des données ou déclenchent un paiement.

6

Passer à Streamable HTTP et sécuriser le serveur

Jusqu'ici, vous avez exécuté le serveur en stdio et peut-être essayé le transport SSE historique. Vous allez maintenant le rendre accessible à distance avec Streamable HTTP et ajouter les protections nécessaires. L'ancien transport HTTP avec SSE est remplacé par Streamable HTTP. Ce dernier peut toujours diffuser des résultats avec SSE, au format text/event-stream, mais sur un point d'accès MCP unique. Conservez temporairement l'ancien accès si des clients en dépendent et préparez leur migration. Les différences principales sont le point d'accès unique, au lieu de routes SSE séparées, et l'utilisation de JSON par défaut. Le client doit annoncer les deux formats dans Accept : application/json et text/event-stream. Le serveur peut répondre par un objet JSON ou par un flux SSE.

En exploitation, Streamable HTTP simplifie le routage et le traitement des données. Son point d'accès unique s'intègre plus facilement aux passerelles et aux outils de CI. Une fois le parcours validé, vous disposez d'une base plus simple à maintenir.

Étapes de migration vers Streamable HTTP

Conservez l'ancien transport SSE uniquement comme solution de repli pendant la transition.

Exposez le point d'accès MCP HTTP unique avec des réponses JSON par défaut.

Vérifiez que l'échange initialize/initialized fonctionne de bout en bout.

Renseignez la nouvelle URL dans la configuration du client et testez avec Inspector.

Organiser le serveur HTTP

Exposez un point d'accès MCP unique au comportement prévisible. Les clients y envoient leurs messages JSON-RPC par POST. Le serveur répond par un objet JSON de type application/json ou par un flux text/event-stream lorsque l'appel nécessite une diffusion progressive. Le principe reste le même : l'échange initialize/initialized précède les appels d'outils.

Considérez la fin de cette initialisation comme la condition d'ouverture de la session, quel que soit le transport.

Avec FastMCP, le lancement ressemble à ceci :

```
if __name__ == "__main__":
    mcp.run(transport="streamable-http")
```

Par défaut, connectez Inspector et les clients à http ://localhost :8000/mcp. Le chemin peut être configurable selon la manière dont l'application est montée.

Contrat du serveur Streamable HTTP

Un point d'accès unique accepte les messages MCP JSON-RPC et renvoie du JSON ou diffuse les résultats.

JSON est le format par défaut ; les réponses SSE permettent une diffusion progressive si nécessaire.

L'échange initialize/initialized reste obligatoire avant les appels d'outils.

L'ancien transport SSE peut être maintenu temporairement pour les clients existants.

Chaque requête est journalisée avec un identifiant de corrélation pour suivre les appels de bout en bout.

Avant même de définir les scopes OAuth, ajoutez quelques protections de base à l'accès HTTP :

— Validez l'en-tête Origin pour les requêtes accessibles depuis un navigateur. Refusez les origines inattendues avec une réponse 403 afin de réduire le risque d'attaques par rebinding DNS contre les services locaux ou internes.
— Pour un serveur strictement local, écoutez sur 127.0.0.1. Exigez une authentification pour tout accès depuis une autre machine.
— Pour un déploiement sur Internet, privilégiez TLS et une passerelle capable de gérer la terminaison TLS, la limitation du débit et l'authentification à l'entrée.

Intégrer la sécurité côté serveur

Pour protéger rapidement le service sans modifier fortement l'infrastructure, commencez côté SDK et serveur. Les SDK MCP permettent d'intégrer des mécanismes d'authentification, notamment OAuth 2.1, afin d'exiger des jetons bearer et de contrôler les scopes avant l'exécution des outils.

Avec Streamable HTTP, l'hôte joint un jeton d'accès à chaque requête, par exemple dans Authorization : Bearer …. Le serveur ou la passerelle doit le valider avant de terminer l'initialisation MCP. Cette validation peut être locale, en vérifiant la signature et les attributs d'un JWT, ou passer par l'introspection du jeton.

Lecture de la figure : Client ID = identifiant du client ; Login screen = écran de connexion ; User + pass = identifiant utilisateur et mot de passe ; Auth code = code d'autorisation ; Exchange auth code for token = échange du code contre un jeton ; Auth server = serveur d'autorisation ; Resource server = serveur de ressources ; Check token / introspect = vérifier le jeton ou effectuer son introspection ; Here's your data = voici les données. Le parcours de droite commence par vérifier si un jeton est disponible, puis s'il est valide. Si aucun jeton valide n'est disponible, un nouveau jeton est demandé : présentation des informations d'authentification à /authorize, puis échange du code contre un jeton à /token. Le serveur de ressources est ensuite appelé avec le jeton via /resource, puis renvoie les données. Yes = oui ; No = non.

Les principales étapes sont les suivantes :

— Activez la prise en charge d'OAuth 2.1 dans votre pile serveur. Validez les jetons bearer et refusez les appels non authentifiés avant de poursuivre l'initialisation.
— Utilisez les scopes pour limiter les capacités accessibles. Les droits de lecture permettent de lister les outils et de lire les ressources autorisées ; les opérations d'écriture ou de paiement exigent des droits plus élevés.
— Placez les informations nécessaires sur les rôles de l'utilisateur dans le jeton pour permettre au serveur de décider rapidement, sans requête supplémentaire.

Cette approche limite les conséquences d'un incident et permet d'ajouter la sécurité sans imposer une nouvelle infrastructure.

Centraliser la sécurité dans une passerelle

Pour appliquer des règles communes à plusieurs services, placez un proxy inverse ou une passerelle d'API devant les serveurs. La passerelle assure la terminaison TLS, valide les jetons, limite le débit et transmet au serveur MCP les requêtes autorisées. Vous centralisez ainsi les règles et réduisez les bibliothèques d'authentification à maintenir dans chaque application.

Points à prévoir :

— Placez le point d'accès MCP Streamable HTTP derrière la passerelle.
— Configurez la validation des JWT et le contrôle des scopes dans les règles de la passerelle.
— Limitez le débit et la taille des requêtes pour protéger les services appelés par les outils.
— Ne transmettez que les attributs du jeton nécessaires au serveur pour limiter les données présentes dans les journaux.

Si vous exploitez plusieurs serveurs MCP ou prévoyez un accès depuis Internet, privilégiez cette centralisation. Sinon, un mécanisme intégré au serveur constitue un premier choix simple et rapide.

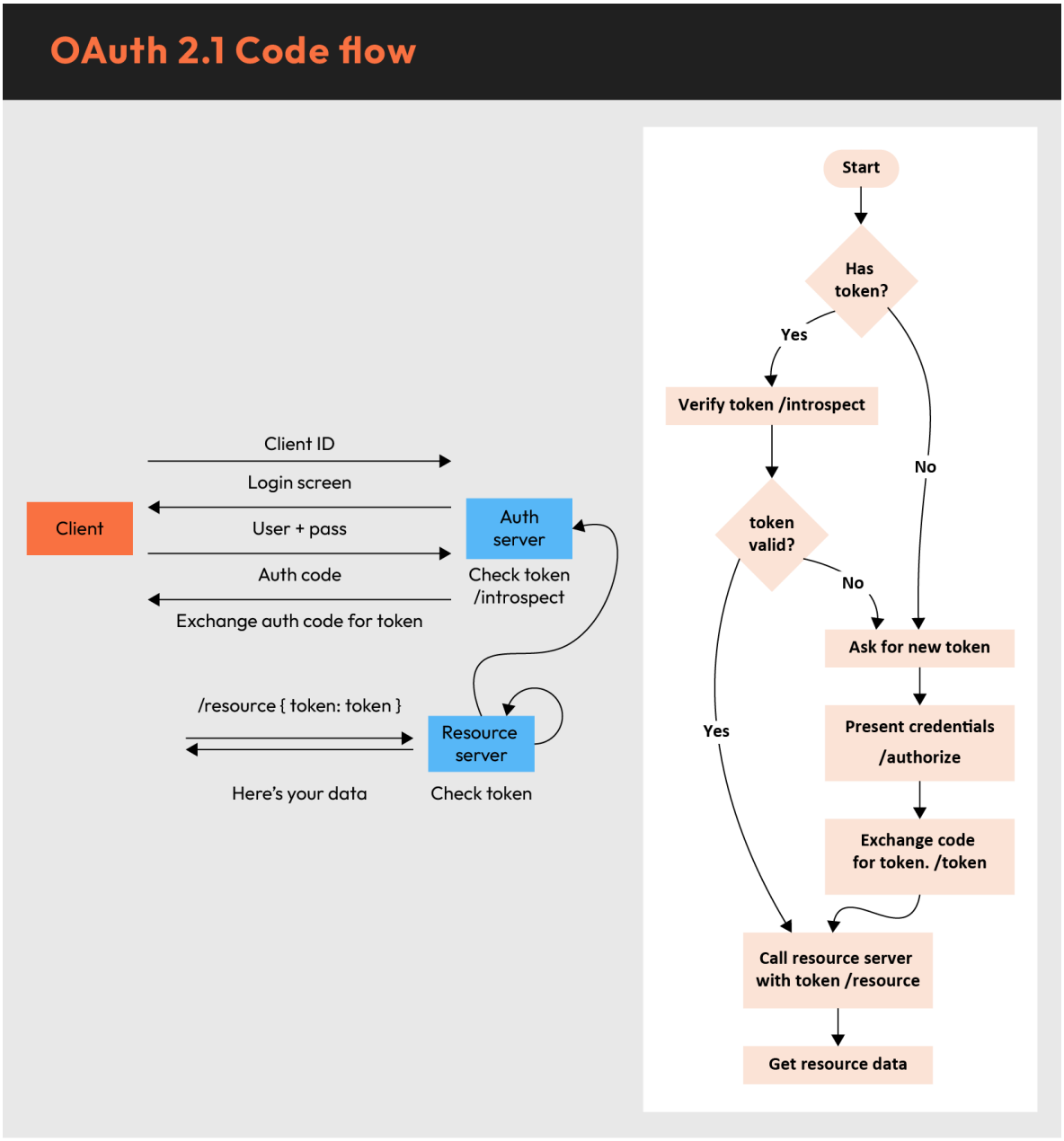

Figure 6.1 : Le flux OAuth 2.1 utilisant un code d'autorisation.

Définir les rôles et les permissions

L'authentification identifie l'appelant. L'autorisation détermine ce qu'il peut faire. Associez les attributs du JWT à des rôles et à des scopes, puis rattachez ces droits à des outils et à des ressources précis.

Voici quelques pratiques utiles :

— Extrayez les attributs de rôle ou de scope du jeton bearer.
— Pour le contrôle d'accès par rôles, ou RBAC, définissez quelques rôles, par exemple reader et operator, et associez-les à des groupes d'outils.
— Protégez les outils sensibles par défaut : leur accès doit exiger un scope élevé explicitement accordé.
— Consignez les décisions d'autorisation avec un identifiant de corrélation pour suivre les appels.

Dans le serveur de référence, les droits de base peuvent donner accès à add et joke. Réservez un rôle plus strict aux outils sensibles qui agissent sur d'autres systèmes.

Préparer le déploiement

Traitez le point d'accès Streamable HTTP comme une API web de production. Stockez les secrets dans des variables d'environnement, jamais dans le code. Ajoutez des contrôles de santé et de disponibilité pour que l'orchestrateur sache quand acheminer les requêtes. Prévoyez une observabilité suffisante pour analyser les appels sans connecter de débogueur au trafic réel. Votre déploiement doit comprendre :

— Les variables d'environnement et les secrets nécessaires à OAuth et aux services appelés.
— Des contrôles de santé et des sondes de disponibilité rapides et sans effets de bord.
— Des points de journalisation indiquant le début et la fin des requêtes, le nom de l'outil, l'identifiant de corrélation et les erreurs.

Vérifiez le parcours complet avant d'ouvrir le pare-feu. Connectez Inspector à l'URL HTTP, listez les outils et effectuez un appel en ligne de commande. Confirmez que l'initialisation et l'authentification fonctionnent, puis testez votre client intégrant un LLM avec la même URL. Exemples de commandes :

— Accès distant en Streamable HTTP avec transport explicite :

```
npx @modelcontextprotocol/inspector-cli https://your-server.example.com/mcp --transport streamable-http --method tools/
```

— Accès distant avec un en-tête d'authentification :

```
npx @modelcontextprotocol/inspector-cli https://your-server.example.com/mcp --transport streamable-http --method tools/
```

Sous Windows, utilisez $env :TOKEN dans PowerShell ou %TOKEN% dans l'invite de commandes pour référencer la variable du jeton.

Avant d'autoriser le trafic réel, transformez ces vérifications en un petit contrôle bloquant de CI. Il doit lister les outils, appeler une ou deux méthodes peu coûteuses et échouer immédiatement si l'initialisation, le schéma, l'authentification ou le transport ne fonctionne pas. Il s'agit d'un test de bon fonctionnement, pas d'un test de charge. Vérifiez les points suivants :

— Les noms d'outils attendus apparaissent et leurs paramètres obligatoires sont toujours présents.
— Les ressources et les modèles de prompt sont également accessibles : ajoutez une lecture avec resources/read et une récupération de prompt avec vérification du contenu produit.
— Pour un outil déterministe comme add(), le résultat correspond exactement à la valeur attendue.
— Pour un résultat variable comme celui de joke(), le format est correct et le contenu n'est pas vide ; le texte exact peut changer.
— Toute erreur d'Inspector fait échouer le contrôle. Conservez stdout et stderr comme artefacts du pipeline.

Les machines de CI doivent disposer de Node.js pour exécuter Inspector. Utilisez npx pour démarrer rapidement ou fixez une version d'Inspector comme dépendance de développement pour rendre les contrôles reproductibles. Avec stdio, la CI doit connaître la commande exacte et ses arguments. Avec Streamable HTTP, elle doit connaître l'URL MCP et les en-têtes d'authentification nécessaires.

Diagnostiquer un échec de CI par couche

Vérifiez d'abord l'initialisation. Si tools/list échoue, la connexion n'a peut-être pas été correctement établie.

Vérifiez ensuite le transport. Avec stdio, Inspector lance le processus. Avec Streamable HTTP, l'URL doit être joignable et l'authentification valide.

Examinez enfin le schéma. Un outil peut apparaître dans le catalogue et échouer si des types ont changé ou si des champs obligatoires manquent.

Les causes les plus courantes sont un environnement d'exécution manquant, un mauvais chemin, un transport incorrect ou un schéma qui a évolué. Corrigez d'abord la plus petite couche défaillante.

À mettre en pratique

— Exposez un point d'accès Streamable HTTP unique et conservez l'ancien transport SSE uniquement comme solution temporaire de compatibilité.
— Activez OAuth 2.1, exigez des jetons bearer et appliquez les scopes minimaux nécessaires.
— Associez les rôles aux outils et refusez par défaut l'accès aux capacités sensibles.
— Pour un service accessible depuis Internet, utilisez une passerelle avec limitation du débit.
— Ajoutez à la CI des commandes Inspector qui listent les outils et réalisent un appel de test.

Avec un accès HTTP protégé et quelques vérifications bloquantes en CI, vous pouvez transformer le serveur d'exercice en une démarche de construction et de déploiement reproductible.

7

Mettre en pratique la démarche de construction et de déploiement

---

Repartez des trois rôles — hôte, client et serveur — et de l'échange d'initialisation. Créez ensuite le projet avec uv, installez le SDK MCP et ses outils en ligne de commande, puis implémentez quelques outils, une ressource et un prompt. Enfin, utilisez l'interface d'Inspector pour une première vérification rapide.

Préparer le projet et les dépendances avec uv

Créez un projet Python avec uv et ajoutez le SDK MCP ainsi que ses commandes CLI. Installez également Node.js pour Inspector. Commencez par un outil et une ressource avant d'élargir le serveur. Définissez des types de paramètres explicites et une description précise.

Effectuez ensuite un premier test dans l'interface d'Inspector : indiquez la commande du serveur et ses arguments, connectez-vous, listez les outils, réalisez un appel et lisez la ressource d'exemple.

Construire une première version en quinze minutes

Créez un projet Python avec uv et installez le SDK MCP avec les outils CLI.

Implémentez un outil avec des paramètres typés et une description claire.

Ajoutez une ressource de contexte statique.

Démarrez le serveur local en stdio.

Ouvrez Inspector, connectez-vous, listez les outils et réalisez un appel.

Intégrer le serveur à un hôte

Raccordez le serveur à une application hôte pour qu'un LLM puisse l'utiliser. Dans VS Code avec Copilot en mode Agent, renseignez .vscode/mcp.json. Dans Claude Desktop, ajoutez la définition correspondante à sa configuration MCP. Dans les deux cas, configurez un serveur stdio avec uv ou python et le chemin exact vers le serveur.

Testez ensuite quelques demandes en mode Agent. Dans notre exemple, demandez une addition et une blague, puis autorisez les appels à add et joke. Le panneau Configure Tools permet d'activer ou de désactiver les outils pendant le débogage de la sélection.

Une fois les essais locaux fiables, passez à HTTP pour rendre le serveur accessible à d'autres utilisateurs. Exposez un point d'accès Streamable HTTP unique et vérifiez l'échange initialize/initialized. Effectuez quelques contrôles de santé avant de continuer. Depuis la CI, utilisez Inspector en ligne de commande pour récupérer une liste d'outils valide à cette URL. Depuis une machine de développement, réalisez aussi un appel. Les erreurs de migration viennent souvent d'un transport incorrect ou d'un mauvais chemin : contrôlez l'URL exacte et relancez les tests. Pour plusieurs serveurs, limitez les contrôles de CI au catalogue de chacun et à un appel simple ; réservez les scénarios complets à une suite de tests d'intégration de niveau supérieur.

Choisir où appliquer la sécurité

Prévoyez la sécurité dès le départ. Deux approches s'offrent à vous : contrôler les accès dans le serveur, avec les mécanismes du SDK, ou placer une passerelle devant le transport HTTP. La première garde les vérifications près du code et s'installe rapidement. La seconde centralise l'authentification, la limitation du débit et le routage.

Inscrivez les rôles ou les scopes dans les jetons et associez-les aux permissions des outils. Commencez avec peu de rôles. Ne placez pas de secrets durables dans le code ou le dépôt ; utilisez les entrées sécurisées de l'hôte ou des variables d'environnement lorsque c'est possible.

Organiser la maintenance et le dépannage

Préparez un court guide d'exploitation pour qu'un autre ingénieur puisse lancer, tester et dépanner le service sans connaître tout le projet. Fournissez un script qui liste les outils et appelle un outil sans risque. Il vérifiera l'initialisation et le routage sans navigateur. Indiquez aussi l'équipe responsable, son canal de contact et l'emplacement des journaux.

Guide de dépannage

— Si Inspector ne se connecte pas, vérifiez Node.js, le chemin du serveur et l'environnement d'exécution.
— Si les outils n'apparaissent pas, vérifiez d'abord initialize/initialized, puis la commande de lancement et le répertoire de travail.
— Si une demande ne déclenche pas l'outil attendu, précisez sa description et les types de ses paramètres.
— Si le service HTTP répond mais que les tests CLI échouent, vérifiez la route Streamable HTTP et assurez-vous que la passerelle transmet le JSON sans le modifier.

Vue d'ensemble de la construction et du déploiement

Parcours de construction

Serveur local en stdio avec un outil et une ressource : Inspector liste les outils et exécute un appel.

Intégration à l'hôte avec mcp.json ou la configuration Claude : une demande déclenche l'outil après autorisation.

Scripts CLI pour tools/list et un appel tools/call sans risque : les contrôles de CI réussissent.

Parcours de déploiement

Déploiement Streamable HTTP : Inspector CLI liste les outils à partir de l'URL.

Maintien temporaire de l'ancien transport SSE si des clients en dépendent.

Application de la sécurité côté serveur ou dans une passerelle : les appels non autorisés sont refusés.

Contrôle minimal en CI : la liste des outils et un appel simple réussissent sur le point d'accès de production.

Étendre le périmètre de votre application

Après avoir livré un premier serveur fiable, élargissez progressivement votre application en combinant d'autres serveurs. Vous pouvez ajouter des serveurs tels que Stripe, GitHub ou Playwright au client ou à l'hôte, puis laisser le LLM choisir leurs outils. Un serveur de fichiers local constitue aussi un moyen simple de fournir du contexte statique au moyen de ressources.

Quelle que soit l'évolution retenue, gardez des choix de transport clairs — stdio en local, Streamable HTTP à distance —, intégrez la sécurité à chaque étape et utilisez Inspector pour vérifier chaque liaison.

À mettre en pratique

Pour prolonger le serveur de référence construit dans ce livre, vous pouvez suivre ces pistes :

— Ajoutez un deuxième serveur MCP à l'hôte et testez une demande impliquant plusieurs serveurs pour vérifier la sélection des outils.
— Ajoutez un serveur de fichiers et lisez une ressource au cours d'un scénario utilisant des outils.
— Déployez le point d'accès Streamable HTTP et appliquez-lui les mêmes contrôles Inspector en ligne de commande.

— Configurez l'authentification côté serveur ou dans une passerelle, puis vérifiez le refus des appels non autorisés.
— Automatisez dans la CI deux contrôles visant les URL de production : tools/list et tools/call avec un outil sans risque.

Vous disposez maintenant d'un parcours allant du développement local au déploiement HTTP sécurisé, avec intégration aux hôtes et contrôles de CI réutilisables à mesure que le projet grandit. Cette base vous permet de livrer plus sereinement et d'étendre le serveur au fil des besoins.

À propos de l'auteur

Ayi NEDJIMI

Auditeur senior en cybersécurité et consultant en intelligence artificielle

Ayi NEDJIMI est auditeur senior en cybersécurité et consultant en intelligence artificielle. Fort de plus de 25 ans d'expérience sur des missions critiques, il accompagne les organisations dans l'évaluation de leurs risques, la sécurité de leurs infrastructures et leurs projets d'IA.

Ancien développeur Microsoft à Redmond, il a travaillé sur le module d'authentification GINA de Windows NT4. Il est également co-auteur de la version française du guide de sécurité Windows NT4 pour la NSA.

À la tête d'Ayi NEDJIMI Consultants, il intervient sur des audits ISO 42001 et ISO 27001, des tests d'intrusion d'infrastructures critiques, des investigations numériques et des missions de conformité NIS2 et AI Act. Ses domaines d'expertise couvrent aussi les environnements cloud, Active Directory, les grands modèles de langage et les architectures RAG.

Expert judiciaire auprès de la Cour d'appel de Paris, il dispose d'une habilitation Confidentiel Défense.

Conférencier en Europe et aux États-Unis, il a formé plus de 10 000 professionnels. Son parcours comprend plus de 100 missions et la publication de plus de 700 articles techniques. Il partage son expertise à travers la formation, ses publications et ses projets consacrés à la cybersécurité et à l'intelligence artificielle.

Retrouver l'auteur

ayinedjimi-consultants.fr · ayi@ayinedjimi-consultants.fr

linkedin.com/in/ayi-nedjimi · github.com/ayinedjimi · orcid.org/0009-0001-5056-3253