Progress®
OpenEdge®

# Use the OpenEdge MCP Server

# Copyright

Visit the following page online to see Progress Software Corporation’s current Product Documentation Copyright Notice/Trademark Legend: https://www.progress.com/legal/documentation-copyright.

**August 2026**

**Product version:** Progress OpenEdge MCP Server 1.1

**Updated: 2026/10/26**

# Table of Contents


## Part I: Introduction and architecture of OpenEdge MCP Server.....11

### Introduction to the OpenEdge MCP Server.....13

Enterprise AI integration with OpenEdge MCP Server.....15
OpenEdge MCP Server terminology.....16
Key features.....18
Production-grade capabilities.....18

### Key differentiators and recommended deployment scenarios.....21

Comparative analysis: OpenEdge MCP Server vs generic MCP solutions.....22
When to deploy the OpenEdge MCP Server.....22

### OpenEdge MCP Server architecture.....25

OpenEdge MCP Server in the AI ecosystem.....25
Core components of the OpenEdge MCP Server architecture.....27
Request lifecycle in OpenEdge MCP Server.....29
OpenEdge MCP Server request flow scenarios.....30
Ensure secure operations in the OpenEdge MCP Server architecture.....33

## Part II: Get started and deployment.....37

### Get started with OpenEdge MCP Server .....39

Deployment components.....39
Supported environments.....41
Python configuration.....43
Install OpenEdge MCP Server.....44
Step 1: Download and unpack the distribution package.....44
Step 2: Run the container image load script.....45
Step 3: Verify your environment.....47
Step 4: Provide OpenAPI specification.....48
Step 5: Generate and start a profile.....49
Step 6: Validate the tool catalog.....55
Enforce security best practices before deployment.....57
Deploy OpenEdge MCP Server to production.....59
Prepare a production-ready export.....60
Run exported bundle on production host.....61

Reconfigure MCP Server without profile regeneration....63
Version verification and troubleshooting....64

## Part III: Upgrade OpenEdge MCP Server....67

Upgrade OpenEdge MCP Server....67
Linux upgrade procedures....68
Step 1: Back up the current deployment....68
Step 2: Perform in-place upgrade....70
Step 3: Validate standard profile upgrade....71
Step 4: Validate profiles with prompts and context enrichment....72
Step 5: Upgrade production exports....72
Step 6: Pin explicit image references (Optional)....74
Step 7: Rollback procedures....75
Windows upgrade procedures....77
Step 1: Back up the current deployment....77
Step 2: Perform in-place upgrade....78
Step 3: Validate standard profile upgrade....78
Step 4: Validate profiles with prompts and context....79
Step 5: Upgrade production exports....79
Step 6: Pin explicit image references (Optional)....81
Step 7: Rollback procedures....82
Post-upgrade best practices....84
Troubleshoot OpenEdge MCP Server upgrade....85

## Part IV: Observability, logging, and troubleshooting....87

## Observability and logging in OpenEdge MCP Server....89

Logging architecture overview....91
Configure logging and related middleware....91
Logging modes and usage guidelines....92
Audit events and sensitive data handling....93
Configure timing middleware....94
Correlation and traceability....95
Integration with log aggregators....95
Debug for rapid issue diagnosis....96
Quick enablement matrix....97
Configure root logging and OpenAPI visibility for debugging....97
Scenario-based troubleshooting....98
Troubleshooting and verification....100
Inspect logs in Kubernetes deployments....101
Configure a minimal debug profile for temporary use....101
Enable targeted diagnostics for invalid parameters....102
Reduce log noise after debugging....102

Add custom loggers for extended debugging....103
Triage checklist before filing issues....103

**Troubleshooting reference ....105**

**Frequently asked questions....109**

**Part V: Operate, secure, and manage the OpenEdge MCP Server....117**

**Use the mcpgen CLI for profile generation and management....119**

Understand profiles and artifacts....120
Manage core commands in mcpgen....120
Create and start a profile....120
List a profile....125
Start, stop, and restart profile containers....125
View container logs....126
Check profile status....126
Profile management and advanced operations....127
Export profiles for production....127
Harden development profiles....128
Generate OpenAPI specification....129
Enable write operations (POST, PUT, DELETE)....130
Rotate service account keys and tokens....132
Service account multi-token management....132
Cleanup and lifecycle management commands....136
Diagnostics and inspection commands....136
Advanced commands....144
Quick start commands to create development profiles....145
Typical workflow and API discovery....146
Error patterns and remedies....149
Clean removal of profile....150
Debug and diagnostics....150

**Operational guidelines and performance best practices for MCP servers....153**

Routine operational tasks....154
Improve system observability....154
Capacity and performance....154
Safe upgrade process and rollback preparation....156
Configuration drift control....156
Prompts lifecycle....157

Security posture enhancements....157
Incident response snapshot....157
Operational runbooks for MCP server....158
Estimate key performance indicators....158

**Configuration reference ....159**

Configuration groups....159
The api group....160
The security group....164
The runtime group....181
The assistant group....193
The observability group....193
Configure environment variable overrides....194
Minimal configuration examples....196
Guidance and common pitfalls ....200

**Security and authentication architecture for MCP server....203**

Threat model and mitigation strategies....204
Dual authentication process....204
JWKS retrieval and caching for JWT verification....205
Scope gating for discovery and execution....206
Protected resource metadata and RFC 9728 compliance....206
Token exchange for downstream services....208
TLS and mTLS for secure communication....208
Response guard strategy....209
Logging hygiene....209
Key and token rotation cadence....209

**Manage prompts and tooling....211**

Configure the prompt directory and size guard....213
Use helper tools for prompt management....213
Secure prompts with scopes....214
Add new prompts....214
Selection strategies for prompts....215
Failure modes and resolutions....215

**OpenAPI visibility and context management....217**

Define goals and guiding principles....218
Visibility and context features ....220
Generate filter examples....222
Provide parameter examples....223
Manage per-tool context....223

Configuration examples for visibility and context features....228
Test visibility and context enrichment features....229
Quick checklist for enabling context features....229

## Export and harden production deployment....231

Export modes....232
Hardening actions....232
Export process....240
Quality, security, and compliance checklist for MCP deployment....240
Run the application in production....243
Scaling considerations....244
Horizontal scaling....244
Vertical scaling....245
Observability in production....247
Security and performance checklist....248
Rotate security artifacts in production....249
Disaster recovery guidelines for MCP servers....250
Common failure modes and resolutions....250

## Rate limiting in OpenEdge MCP Server....251

Implement rate limiting....252
Configuration keys for rate limiting....252
Recommended rate limiting values for different environments....253
Monitor rate limiting through observability and audit fields....254
Safe rollout strategy for rate limiting....255
Alerting suggestions for rate limiting....255
Tune and troubleshoot rate limiting....255

## Upgrade process and key or certificate rotation....257

Version and upgrade principles....258
Pre-upgrade checklist and upgrade procedure....258
Service account key rotation....259
TLS and mTLS certificate rotation....260
Token renewal and revocation....262
Patterns to minimize downtime....262
Observability hooks during rotation....262
Rollback decision matrix....263
Automation suggestions....263

## Part VI: OpenAPI specification and advanced configuration....265

**OpenAPI specification quality guidelines.....267**
Core principles for OpenAPI specifications.....268
Request body guidelines.....268
Define response schemas.....269
Component schema hygiene.....270
Define titles and descriptions for schemas and operations.....271
Ensure clarity in parameter definitions.....272
Error conventions.....273
Versioning and deprecation guidelines.....273
Hand-editing checklist before commit.....273
Common pitfalls in API specifications.....274
Minimal example for API specification.....275

# I

# Introduction and architecture of OpenEdge MCP Server

For details, see the following topics:

- Introduction to the OpenEdge MCP Server
- Key differentiators and recommended deployment scenarios
- OpenEdge MCP Server architecture

# 1

# Introduction to the OpenEdge MCP Server

The Progress® OpenEdge® MCP Server is a production-grade implementation of the Model Context Protocol (MCP) open standard, which is designed to securely expose OpenEdge business logic and data to AI workflows.

## What this guide contains

This guide is designed to be your comprehensive resource for understanding, deploying, and operating the OpenEdge MCP Server in enterprise environments. It provides a structured pathway from foundational concepts to advanced operational practices, ensuring you can confidently integrate AI workflows with OpenEdge business systems.

In this guide, you will find:

- **Introduction and architecture of OpenEdge MCP Server—**Explore the challenges of AI integration with enterprise data, the requirements for secure and compliant workflows, and how OpenEdge MCP Server, built on the Model Context Protocol (MCP), addresses these needs. This segment introduces the architecture of the OpenEdge MCP Server, highlights its key differentiators, and outlines recommended deployment scenarios.
- **Get started and deployment—**Step-by-step instructions for installing, configuring, and deploying the OpenEdge MCP Server. This includes environment prerequisites, supported platforms, and workflows for moving from development to production.
- **Upgrade OpenEdge MCP Server—**Comprehensive platform-specific upgrade workflows for transitioning between OpenEdge MCP Server versions. This section includes backup and validation procedures, image management strategies, configuration compatibility guidance, and rollback contingencies to ensure safe, validated upgrades with full functional verification.
- **Observability, logging, and troubleshooting—**Tools and techniques for monitoring system health, auditing events, and diagnosing issues. This segment also includes a comprehensive FAQ section to address common questions and provide quick solutions to typical challenges.
- **Operate, secure, and manage the OpenEdge MCP Server—**A detailed reference on profile management with `mcpgen` CLI, operational best practices, security, prompt and tooling management, OpenAPI visibility, rate limiting, upgrades, and key or certificate rotation for maintaining a secure and efficient OpenEdge MCP Server deployment.
- **OpenAPI specification and advanced configuration—**A comprehensive reference for best practices in designing high-quality OpenAPI specifications, advanced configuration, and strategies to ensure clarity, consistency, and compliance in your API documentation.

---

**Disclaimer:** This document provides general security guidance for remote system configurations including the MCP Server. The documented techniques must be fully vetted and evaluated by the customer for completeness and viability in their unique deployment scenarios.

---

For details, see the following topics:


- Enterprise AI integration with OpenEdge MCP Server
- OpenEdge MCP Server terminology
- Key features
- Production-grade capabilities

# Enterprise AI integration with OpenEdge MCP Server

## Enterprise challenges in AI integration

Traditional AI workflows often struggle to access structured business data securely, creating challenges for organizations that want to use AI for operational tasks. While unstructured data such as documents, emails, and logs can be easily processed using embeddings and vector stores, structured business data remains a significant hurdle. Orders, inventory, customer records, and transactions stored in OpenEdge databases and governed by business logic are not readily accessible to AI workflows. Without any secure, standardized mechanism that allows AI agents to execute business operations and not just read data while maintaining compliance and governance, exposing sensitive business logic introduces risk and violates enterprise security policies.

## Essential requirements for AI workflow integration

Organizations running OpenEdge applications require AI workflows that can:

- Make intelligent decisions using proprietary business data and rules.
- Execute business operations through natural language interactions.
- Maintain data sovereignty and security compliance in on-premises or private cloud environments.
- Audit and govern AI interactions with business systems to ensure accountability.

## The foundation - Model Context Protocol

The Model Context Protocol (MCP) defines a standardized interface that enables Large Language Models (LLMs) to interact with external enterprise systems in a secure and predictable manner. MCP provides the following capabilities:

- **Tool discovery**—LLMs can identify available business operations through structured metadata exposed by the MCP Server.
- **Context awareness**—Each tool includes detailed information such as parameters, schemas, and business entity types, allowing LLMs to make informed decisions before execution.
- **Secure execution**—Operations are performed under strict authentication and authorization controls to maintain compliance and protect sensitive data.
- **Structured responses**—Responses are returned in a consistent, machine-readable format, enabling LLMs to process results and drive subsequent actions effectively.

## The solution - OpenEdge MCP Server

The OpenEdge MCP Server addresses the challenges in AI integration by transforming an OpenAPI specifications and optional prompt assets into AI-accessible tools. This process makes each API operation discoverable and usable by AI workflows, without requiring manual coding, enabling secure and efficient interaction with enterprise APIs.

The OpenEdge MCP Server acts as a bridge between AI agents and OpenEdge services, exposing APIs as secure, discoverable tools. The OpenEdge MCP Server connects to PAS for OpenEdge services through REST and WEB interfaces, enabling seamless communication between OpenEdge applications and MCP components. The following examples outline how AI-enabled OpenEdge workflows can execute business operations through natural language requests:

| Business operation | Natural language request example |
|---|---|
| Order analysis | *Show me orders from the last 30 days with delivery delays and suggest remediation actions.* |
| Customer onboarding | *Create a new customer account and set up their initial order based on our standard package.* |
| Inventory optimization | *Analyze inventory levels across warehouses and recommend restock priorities.* |

For detailed information about the architecture and core components of the OpenEdge MCP Server, see OpenEdge MCP Server architecture on page 25.

# OpenEdge MCP Server terminology

This section provides a comprehensive glossary of terms related to the OpenEdge MCP Server that you will see throughout this guide.

- **MCP tool**—A secure and discoverable external service that allows AI agents like LLMs to invoke specific business operations exposed by an MCP Server. Each tool corresponds to one OpenAPI operation, such as `GET /orders` or `POST /customers`, and includes metadata for safe execution.

  For example, a database query tool for inventory levels enables AI models to fetch real-time inventory data from business systems using the MCP protocol. This tool allows secure, managed access to database queries, supporting operational decisions, and inventory management.

- **Tool generation**—Each OpenAPI operation is converted into a distinct MCP tool, ensuring one-to-one mapping between API operations and MCP tools. In addition, helper utilities are generated to expose schemas, tags, and prompts for better discoverability and usability. This approach eliminates manual wiring and promotes consistency across tools.

- **Prompt**—A server-provided, Markdown-based asset that guides AI responses and can coordinate multiple tools for complex workflows. It is scope-filtered to ensure visibility only to authorized users and size-guarded to prevent excessive memory consumption or injection risks.

- **Scope**—Named permissions that control both the discovery and invocation of MCP tools. They align with the configured security model, enforcing least privilege by gating access to tools and prompts based on assigned scopes.

- **Routing map**—Define which HTTP methods and paths are permitted or restricted. They use INCLUDE and EXCLUDE directives to enforce routing policies, such as blocking write operations by default unless explicitly allowed.

- **Response guard**—A safety mechanism that enforces limits on response size and array length to ensure that the responses sent back to the client stay within the defined boundaries. It operates in configurable modes, such as `block`, `trim`, or `warn` to prevent runaway responses and maintain predictable behavior during tool execution.

- **Profile**—Reproducible environment bundles generated by the `mcpgen` utility. Each profile includes configuration files, scripts, certificates, and other runtime artifacts required to start and manage an MCP Server instance. Profiles ensure deterministic deployments across environments.

- **Export and hardening**—Create deployable artifacts from development profiles. These artifacts are immutable and include pinned image digests. Hardening steps tighten security posture before production deployment.

- **MCP (Model Context Protocol)**—A standardized interaction model between agents and servers. It enables AI agents to discover, understand, and safely invoke tools exposed by the OpenEdge MCP Server, ensuring secure and structured communication.

- **Service account JSON Web Token (JWT)**—A trusted credential issued to the OpenEdge MCP Server, enabling privileged evaluation of operations.

- **JSON Web Key Set (JWKS)**—A standardized format for representing a set of public cryptographic keys in JSON. JWKS is commonly used to publish the keys required to verify the signatures of JWTs. The OpenEdge MCP Server uses JWKS to securely verify authentication tokens (JWTs) presented by users or service accounts. The server is configured with a `jwks_uri`, which is a URL endpoint provided by an identity provider, such as Azure AD, Keycloak, or AWS Cognito. This endpoint publishes the current set of public keys that the MCP Server retrieves and uses to verify the authenticity of incoming JWTs.

- **User token**—Represents the end-user identity that is passed through or exchanged for downstream calls. It ensures that operations are executed under the correct user context, supporting fine-grained authorization.

- **OpenAPI specification**—a widely adopted standard for describing RESTful APIs in a machine-readable format, such as JSON or YAML. It provides a structured way to define the endpoints, request and response formats, authentication methods, and other details of an API.

# Key features

The OpenEdge MCP Server delivers enterprise-grade capabilities designed for secure and predictable AI integration. Some of the key features include:

- **Zero manual tool wiring**—Automatically converts OpenAPI specifications into a tool catalog at startup.
- **Safe by default**—Non-GET HTTP methods are excluded unless explicitly enabled. This restriction is designed to minimize the risk of unintended data modifications by MCP Clients. It achieves this by disallowing the use of HTTP methods commonly associated with data-altering operations, specifically PUT, POST, and DELETE.
- **Dual authentication**—Combines service account and user token with scope-based gating.
- **Secure discovery**—Tools and prompts are visible only when the caller holds the required scopes.
- **Built-In guardrails**—Enforces response size limits, array length checks, and optional parameter validation.
- **Deterministic profiles**—The `mcpgen` command line interface (CLI) produces directories containing configuration, certificates, tokens, and scripts. It simplifies profile generation, configuration management, and lifecycle operations for OpenEdge MCP environments. For more information, see Use the mcpgen CLI for profile generation and management on page 119.
- **Enterprise transport**—Supports TLS and mTLS for inbound connections and configurable downstream TLS/mTLS.
- **Operational clarity**—Provides structured inventory logging, correlation IDs, and optional rate limiting.
- **Hardened distribution**—Enables export and digest pinning for production or air-gapped deployments.
- **Extensible but opinionated**—Clear configuration groups reduce hidden coupling and environment sprawl.

# Production-grade capabilities

The following dimensions define the production-grade nature of the OpenEdge MCP Server:

| Dimension | Built-In capability |
|---|---|
| Least privilege | • All routes default to read-only.<br>• Write operations require explicit enablement. |
| Authorization model | • Dual authentication combines a service account and user token.<br>• Optional support is available for JSON Web Key Set (JWKS) rotation to ensure key agility. JWKS rotation ensures secure AI tool integration and continuous availability. |
| Discovery security | Scope-based filtering is available for tools and prompts, so users only see what their token allows. |
| Response safety | A centralized guard enforces size and item limit. |
| Determinism | Profiles generated by `mcpgen` are versioned and reproducible, ensuring consistent deployments. |
| Transport hardening | TLS and mTLS are enforced for inbound traffic, with downstream connections reusing the configured TLS settings for trust continuity. |
| Token hygiene | Service account credentials are stripped before forwarding downstream, reducing leakage risk. |
| Observability | Startup inventory and structured logging hooks provide visibility without exposing secrets. |
| Extensibility | Built-in helper tools allow schema, tag, and prompt management without writing custom code. |
| Deployment | Hardened export pipeline includes rotation scripts for keys and certificates to maintain security posture. |

# 2

# Key differentiators and recommended deployment scenarios

This section highlights what makes the OpenEdge MCP Server unique compared to generic MCP solutions. Through a comparative analysis, you learn how architecture and operational features of the OpenEdge MCP Server set it apart in regulated environments. Additionally, clear guidelines are provided to help you determine when deploying OpenEdge MCP Server is the optimal choice and when alternative approaches may be more suitable for your organizational needs.

For details, see the following topics:

- Comparative analysis: OpenEdge MCP Server vs generic MCP solutions
- When to deploy the OpenEdge MCP Server

# Comparative analysis: OpenEdge MCP Server vs generic MCP solutions

Unlike generic MCP servers, the OpenEdge MCP Server is engineered with enterprise-grade security, deterministic workflows, and integrated guardrails that ensure predictable, hardened yet safe AI tool deployment. It provides the needed protection for your application's business logic and data. The following table outlines the differences between the OpenEdge MCP Server and generic MCP solutions:

| Feature | OpenEdge MCP Server | Generic MCP server |
|---|---|---|
| **Secure discovery** | Gates tools and prompts by scopes and does not list them universally. | Typically lists all tools without scope-gating |
| **Dual credential strategy** | Separates service account trust from end-user identity and merges only what downstream needs. | Often uses a single credential or less strict separation |
| **Opinionated defaults** | Blocks write operations unless explicitly enabled | May allow writes by default |
| **Built-In hardening workflow** | Integrated export and hardening steps are a first-class feature and not ad-hoc scripts | Usually relies on ad-hoc scripts or manual steps |
| **Deterministic profiles** | A single command reliably rebuilds the entire environment, including certificates, tokens, and configuration, ensuring consistency every time. | Profiles may vary and are less reproducible |
| **Guardrail stack** | Response and parameter validation are integrated from the start. | Often added later as middleware |
| **Prompt safety** | Size guard and scope-gating prevent large prompt injection | Limited or no prompt safety mechanisms |
| **Minimal runtime surprises** | Parses specification only once, eliminating any risk of uncontrolled dynamic code generation. | May allow dynamic runtime code generation |

# When to deploy the OpenEdge MCP Server

Choose OpenEdge MCP Server when you need to:

- Expose existing OpenEdge or REST APIs to LLM agents rapidly with strong safety controls.
- Enforce structured discovery and execution by applying scope-based routing and disciplined policies from the start.
- Deliver reproducible, attestable MCP surfaces ideal for regulated or multi-tenant environments requiring predictable deployments.
- Operate under strict security requirements by supporting TLS/mTLS, token rotation, and JWKS for hardened transport and identity assurance.

Avoid using OpenEdge MCP Server if:

- You require dynamic code or tool generation beyond the OpenAPI and prompt model.
- You need a full policy engine. OpenEdge MCPServer provides scopes and routing only.
- You expect the server to issue identity tokens. Token management must be delegated to your identity provider.

# 3

# OpenEdge MCP Server architecture

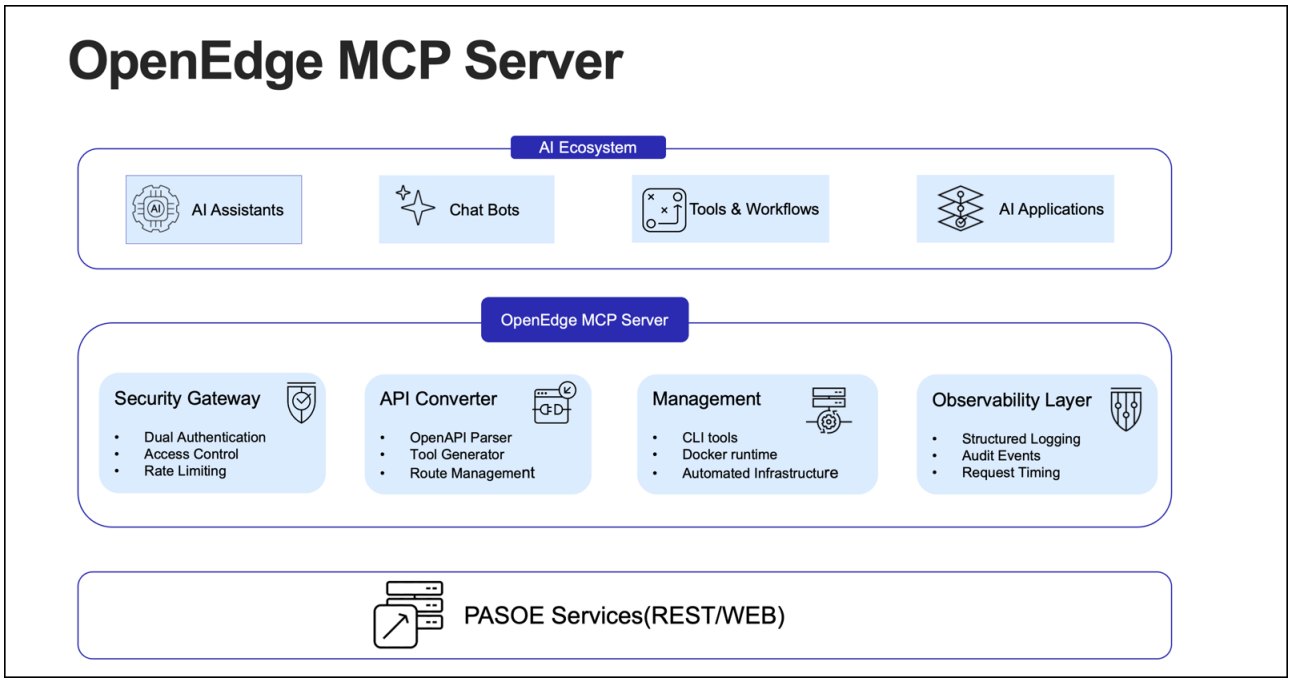
The OpenEdge MCP Server architecture defines how AI-driven workflows securely interact with OpenEdge business logic and APIs. It combines protocol-level controls, identity management, and operational safeguards to ensure that natural language requests can be translated into authorized business operations without compromising security or compliance. This architecture is designed for enterprise environments where trust boundaries, extensibility, and governance are vital.

For details, see the following topics:

- OpenEdge MCP Server in the AI ecosystem
- Core components of the OpenEdge MCP Server architecture
- Request lifecycle in OpenEdge MCP Server
- OpenEdge MCP Server request flow scenarios
- Ensure secure operations in the OpenEdge MCP Server architecture

## OpenEdge MCP Server in the AI ecosystem

The OpenEdge MCP Server sits between the AI ecosystem and PAS for OpenEdge services, acting as a secure bridge that exposes OpenEdge APIs to AI workflows.




It supports an AI ecosystem that includes assistants, chatbots, tools, workflows, and AI applications. AI assistants and agents typically connect through an MCP Client, which communicates with the MCP server. Through this integration, the OpenEdge MCP Server delivers intelligent automation and conversational experiences.

The OpenEdge MCP Server consists of four main layers that ensure secure, scalable, and observable operations:

- Security gateway for dual authentication, access control, and rate limiting.
- API converter for OpenAPI parsing, tool generation, and route management.
- Management layer featuring CLI tools, Docker runtime, and automated infrastructure.
- Observability layer for structured logging, audit events, and request timing.

The OpenEdge MCP Server connects to PAS for OpenEdge services through REST and WEB interfaces, enabling seamless communication between OpenEdge applications and MCP components.

It exposes essential OpenEdge components as secure, discoverable tools for AI workflows. These tools provide a structured interface that enables natural language-driven operations. The following OpenEdge components are exposed:

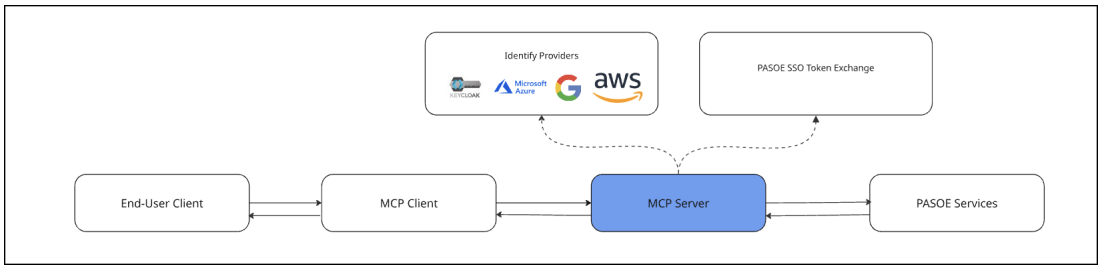
- ABL business logic from Progress Application Server (PAS) for OpenEdge.
- REST and Web APIs defined in OpenAPI specifications.
- OpenEdge Business Entities as discoverable and callable tools.

Each MCP tool includes attributes that maintain security, enforce compliance, and support reliable execution, such as:

- Descriptions, parameter schemas, and entity tags for context-aware execution.
- Scope-based access control to enforce least privilege.
- Safe execution and predictable responses.
- Capturing invocation details for compliance and governance.

# Core components of the OpenEdge MCP Server architecture

Here is an overview of the core components in an AI ecosystem and their interactions within the OpenEdge MCP Server architecture.




- **End-User Client**—The interface where tasks originate. It can be a chat application, web interface, or command-line tool. This client may embed an MCP Client or connect to one externally. It provides user credentials that authorize tool invocations.
- **MCP Client (through an AI Agent)**—The MCP Client communicates with the MCP Server using the Model Context Protocol. It holds two credentials:
  - A service account credential assigned by the server.
  - A user token or JWT representing the end-user identity.

  The MCP Client discovers available tools, selects prompts, and invokes operations on behalf of the end-user. In simple deployments, the MCP Client and end-user client run in the same process. In multi-tier setups, they operate separately. Examples of MCP clients include VS Code extension, Windsurf, custom web chat UI, web dashboard application, and standalone desktop app.
- **MCP Server**—The MCP Server is the central component of this architecture. It converts OpenAPI specifications and optional prompts into scoped tools. Key responsibilities include:
  - Validating both service account and user credentials ensuring dual authentication.
  - Applying scope-based access control.
  - Enforcing routing rules, which are read-only by default.
  - Executing downstream HTTP calls with guardrails for size and performance.

  The MCP Server never forwards its service account credential to downstream systems. When configured with OAuth2 endpoints, it acts as an authorization server, delegating authentication to external Identity Providers and managing OAuth2 protocol flows, including authorization code exchange, proof key for code exchange (PKCE) validation, and token issuance.
- **Identity Provider (IdP)**—External authentication systems, such as Keycloak, Azure AD, AWS Cognito, or Google handle user authentication. The MCP Server delegates login and consent to these providers, which issue OAuth2/OIDC tokens. The MCP Server validates these tokens and may exchange them for downstream usage.
- **PAS for OpenEdge Single Sign-On (SSO) token exchange**—Specifies an optional identity or token exchange endpoint.
  - **When token exchange is configured**—The MCP Server exchanges the inbound user credential for a PAS for OpenEdge-scoped token. This token includes audience, claims, or lifetimes tailored for backend services. If token exchange fails, the request is denied.
  - **When token exchange is not configured**—This component is bypassed.
- **PAS for OpenEdge services**—Specify the OpenEdge or PAS for OpenEdge REST, or other backend services that implement business logic. They receive only the end-user credential or the exchanged token. Responses pass through MCP Server guardrails before reaching the MCP Client.

# Request lifecycle in OpenEdge MCP Server

The following steps outline how a user request is processed from initiation to response within the OpenEdge MCP Server architecture:

1. The end-user interacts with an application, for example an IDE or a chat interface through an AI Agent, which delegates the user query to the MCP Client.

   For example, the end-user enters a natural language request, such as: *“Show me all the available inventory.”*

2. The MCP Client authenticates to the MCP Server using both service account and user credentials. When operating in OAuth2 mode, the MCP Server functions as an authorization server. It delegates user authentication to external Identity Providers using OAuth2 flows, including the authorization code process with PKCE for enhanced security. During initialization, both parties declare their supported capabilities:

   - **Tools** – Functions the server can execute
   - **Resources** – Data the server can provide
   - **Prompts** – Templates offered by the server that guide AI responses and can coordinate multiple tools for complex workflows

   The MCP Server lists tools and prompts allowed by the scope of the user. Once initialized, the connection remains persistent. Both, MCP Client and Server are responsible for handling errors and maintaining the connection state.

3. The MCP Client forwards the query and server information to the LLM, which analyzes the request and recommends the appropriate MCP tools for execution.
4. The MCP Client executes the recommended tools with necessary parameters on the MCP Server.
5. On invocation, the MCP Server optionally exchanges tokens with PAS for OpenEdge SSO.
6. The MCP Server calls PAS for OpenEdge Services with sanitized headers.
7. Response guard enforces limits before returning results to the MCP Client.
8. The MCP Client returns the tool results to the AI Agent, and the LLM incorporates the new information into its reasoning. The LLM may decide to call additional MCP tools if needed, repeating steps 4–7 as necessary.
9. The LLM generates the final response using all available contexts.
10. The AI Agent presents the LLM's final response to the end-user.

    For example, as requested by the end-user in step 1, the inventory summary may appear as:

```
Inventory Summary:

Raw Materials: 3,890 units
Work in Progress: 246 units
Finished Goods: 585 units

Low Stock Alerts: Aluminum Rods, Model X-300

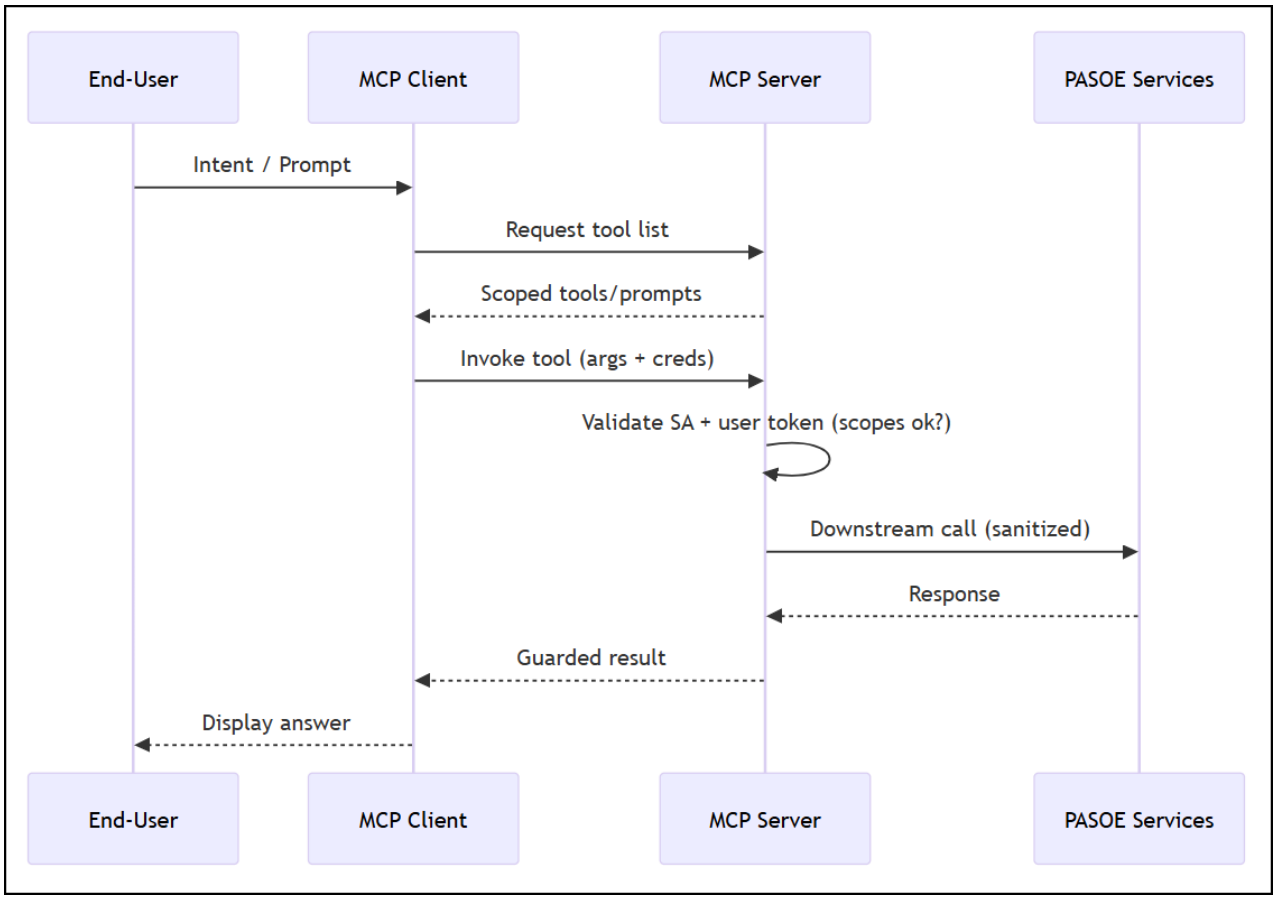
Total Value: $2.4M
Last Updated: Sep 15, 14:45
```

# OpenEdge MCP Server request flow scenarios

This section explains how the OpenEdge MCP Server processes requests under different conditions. Each sequence diagram illustrates the interaction between components and the security measures applied during the request lifecycle.

## Normal request flow

The following image represents a standard successful request flow.

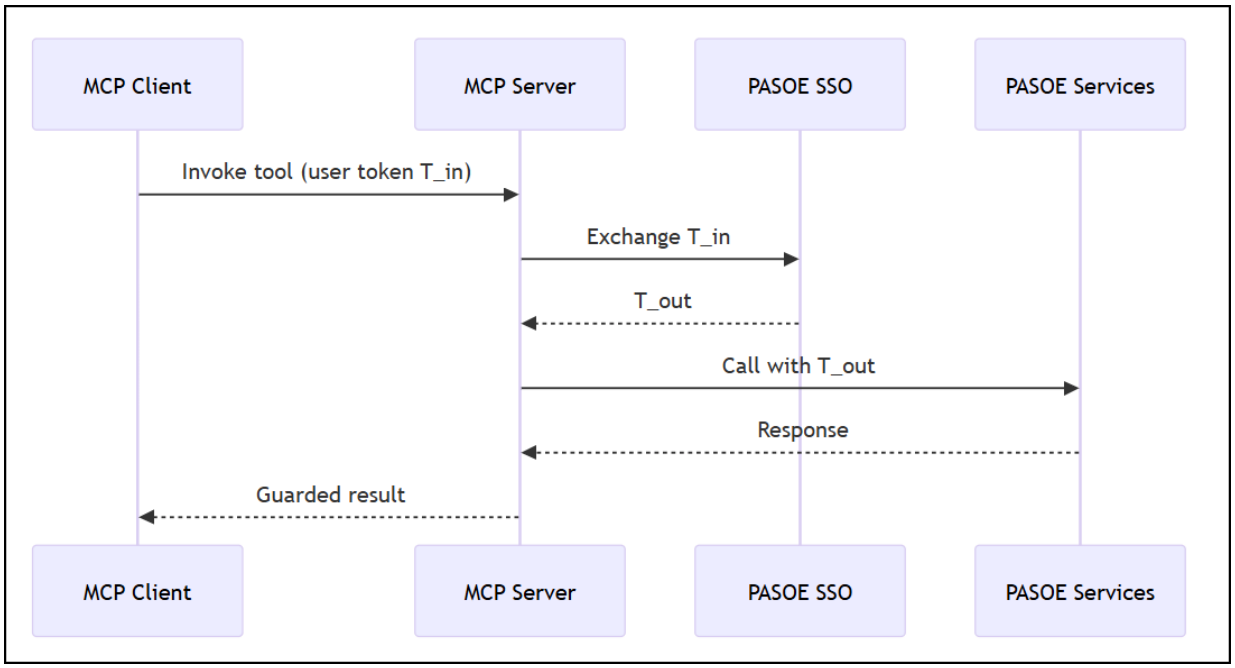



It includes:

- Tool discovery by the MCP Client.
- Invocation of a selected tool.
- Dual authentication and scope validation.
- Guarded downstream call to PAS for OpenEdge services.
- Return of a bounded response to the client.

## Token exchange variant

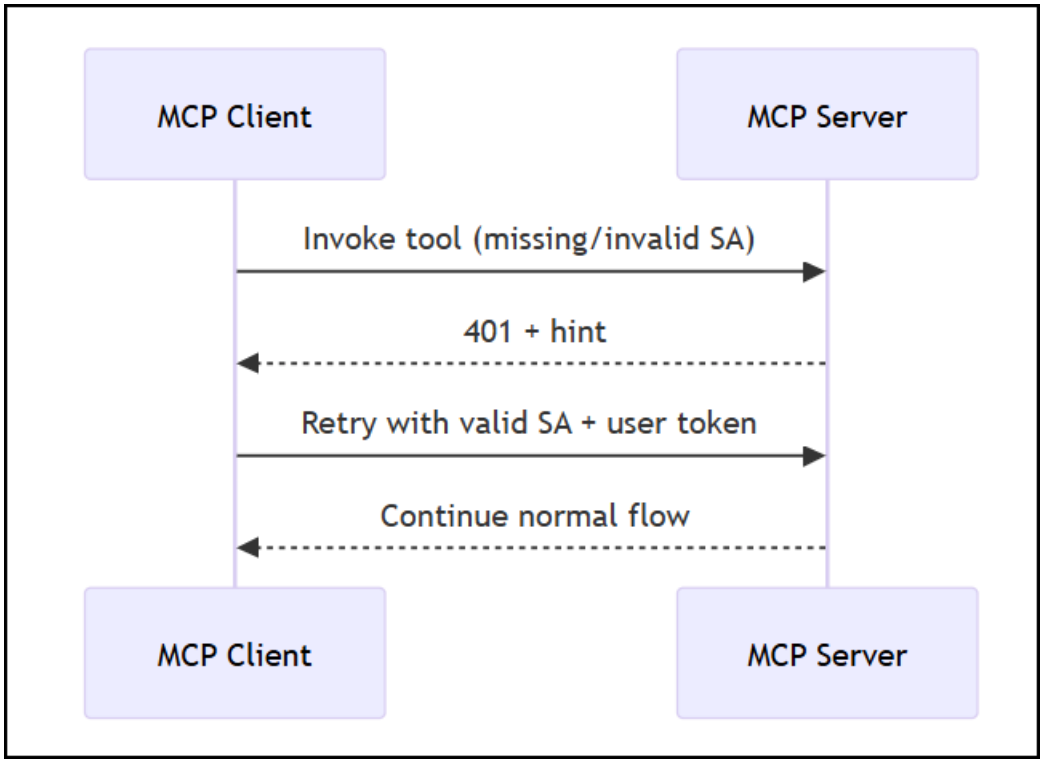
This scenario follows the normal execution path but introduces a token exchange step.




The MCP Server issues a scoped backend token instead of passing the original user token downstream. This approach improves isolation and reduces the credential blast radius by enforcing tighter token lifetimes and scopes.

## Authentication failure and retry

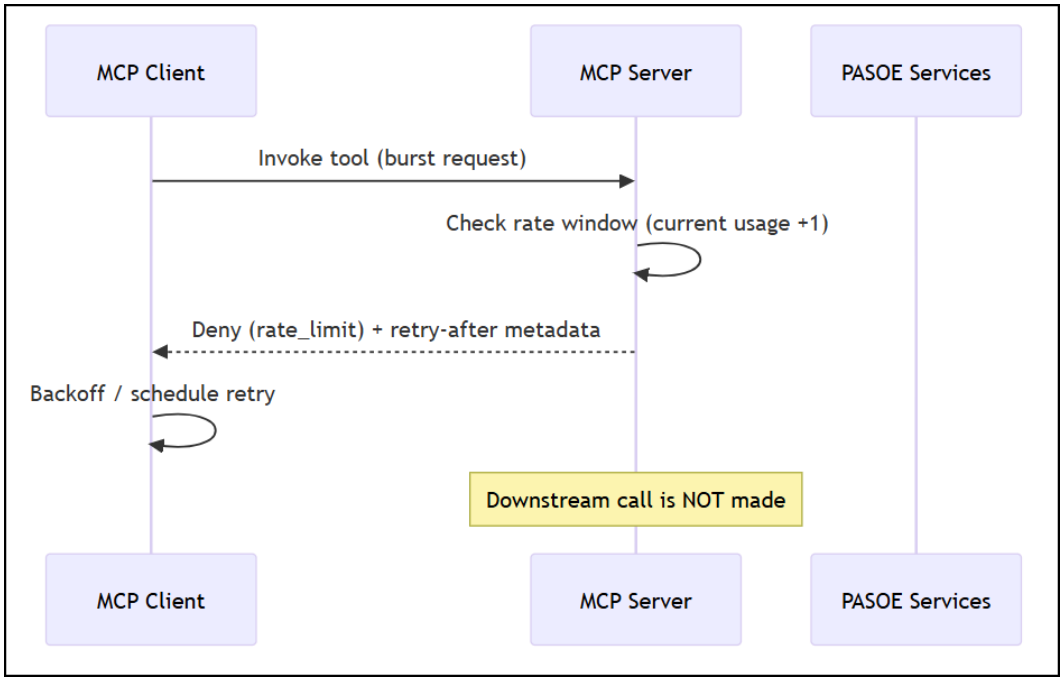
This scenario demonstrates fail-closed behavior, which is a security best practice: instead of allowing a fallback or partial access, the MCP Server blocks the operation and requires the user to retry with valid credentials.



- The request fails if the service account credential is missing or invalid.
- Recovery occurs immediately after a valid service account token is provided.

## Rate limit denial

This scenario shows how the MCP Server enforces throughput policies.




The MCP server:

- Rejects an over-quota request before any downstream call.
- Returns metadata so that the client can apply backoff strategies.

## Token exchange: With and without

The following comparison table highlights the differences between using token exchange and passing the original user token:

- **Without exchange**—The original user token is sent downstream.
- **With exchange:**—The MCP Server issues a scoped token with controlled lifetime and claims.

**Table 1: Token Exchange: With vs Without**

| Aspect | Without Exchange | With Exchange |
|---|---|---|
| When used | Direct user token acceptable by PAS for OpenEdge services | Backend requires different audience, specific claims, or shorter time to live (TTL). |
| Extra network round trip | No additional network call is needed | Adds one call to PAS for OpenEdge SSO endpoint per exchange. This step is often optimized through token caching based on TTL. |

| Aspect | Without Exchange | With Exchange |
|---|---|---|
| Credential sent downstream | The original user token is forwarded to PAS for OpenEdge Services. | The MCP server issues a scoped exchanged token (**T_out**), which is sent downstream to ensure tighter control. |
| Token lifetime and scope control | Lifetime and scope are determined solely by the upstream Identity Provider. | The MCP server enforces stricter limits on token lifetime, audience, and scopes for backend usage. |
| Failure mode | The downstream call proceeds unless the token is invalid. | Exchange failure aborts request (fail closed), preventing fallback to the original user token. |
| Security benefit | Provides simplicity and lower latency because no additional configuration or network round trip is required. | Enhances security by reducing the blast radius and preventing exposure of raw user tokens with excessive scopes. |
| Operational complexity | Lower complexity since no token exchange configuration is needed. | Higher complexity due to additional setup for token exchange endpoints, certificate authority trust, and retry handling. |
| Recommended use | Suitable for development environments or deployments with homogeneous trust boundaries. | Recommended for production environments that require stricter backend security or multi-tenant isolation. |

# Ensure secure operations in the OpenEdge MCP Server architecture

This section outlines the security guarantees provided by the OpenEdge MCP Server and the recommended best practices for maintaining robust security posture. It explains the default protections built into the architecture and offers guidance on encryption, authentication, token management, and operational hardening to minimize risk and ensure compliance.

## Security guarantees

These guarantees form the foundation of the MCP Server security model, minimizing risk and enforcing least privilege:

- Service account credentials are never sent to PAS for OpenEdge services.
- Non-GET operations remain blocked unless explicitly enabled, enforcing least privilege.
- Token exchange produces short-lived, scope-limited tokens.
- Response and rate limit guards protect performance and cost.

## Recommended security posture and best practices

To maintain a strong security posture and reduce the risk of unauthorized access or lateral movement, follow these best practices:

| Best practice | Details |
|---|---|
| Encrypt all connections | • Use TLS for every communication hop:<br>  • Client → MCP Client (if remote)<br>  • MCP Client → MCP Server<br>  • MCP Server → PAS for OpenEdge SSO<br>  • MCP Server → PAS for OpenEdge Services<br>• Enable mutual TLS (mTLS) wherever both endpoints are under your control:<br>  • At minimum: MCP Client → MCP Server<br>  • Optionally, PAS for OpenEdge SSO and PAS for OpenEdge Services if they support client certificate validation<br>This best practice ensures confidentiality, integrity, and authenticated channels, reducing the risk of header or token interception. |
| Prefer token exchange over direct token pass-through | • Allow the MCP Server to issue backend tokens so downstream services trust it as the controlled issuer.<br>• Use short-lived, scope-limited tokens to minimize exposure compared to long-lived generic user tokens.<br>• If token exchange fails, deny the request rather than falling back to the original user token. |
| Enforce dual authentication and scope-gating | • Keep service account verification enabled to establish a stable server identity separate from the end-user.<br>• Apply scope filtering and default read-only routing to ensure only authorized tools are visible and executable. |
| Rotate cryptographic material regularly | • Rotate the following at the recommended cadence of 30-90 days, based on sensitivity:<br>  • Service account keys<br>  • Token signing keys (if applicable)<br>  • TLS/mTLS certificates |

| Best practice | Details |
|---|---|
| Harden TLS configuration | • Use modern cipher suites and disable legacy protocols, such as TLS 1.0/1.1.<br>• Validate server hostname and CA chain for PAS for OpenEdge SSO and PAS for OpenEdge Services.<br>• Pin CA roots if allowed by organizational policy. |
| Minimize token scope and lifetime | • Issue tokens with only the required scopes and claims.<br>• Set short TTL: minutes to a few hours.<br>• Reject or re-exchange tokens older than your policy threshold. |
| Monitor and alert | • Track:<br>  • Rate limit denials<br>  • Authentication failures<br>  • Token exchange errors<br>• Treat unusual spikes as potential abuse.<br>• Log structured events (excluding secrets) to central Security Information and Event Management (SIEM) for anomaly detection. |
| Fallback policy | • If token exchange is unavailable, deny the request explicitly.<br>• Do not silently forward raw user tokens, as this weakens security. |

II

# Get started and deployment

For details, see the following topics:

- Get started with OpenEdge MCP Server

# 4

# Get started with OpenEdge MCP Server

This section provides a complete workflow to install and configure the OpenEdge MCP Server using the provided distribution package, and how to prepare a secure, production-ready deployment. It is intended for developers and administrators who need to securely expose OpenEdge APIs to AI workflows using the Model Context Protocol (MCP).

For details, see the following topics:


- Deployment components
- Supported environments
- Install OpenEdge MCP Server
- Enforce security best practices before deployment
- Deploy OpenEdge MCP Server to production
- Reconfigure MCP Server without profile regeneration
- Version verification and troubleshooting


## Deployment components

OpenEdge MCP Server leverages a containerized approach for deployment, which uses three distinct components. Understanding these components early helps you interpret the commands that follow.

## Component overview

The following components are used for the OpenEdge MCP Server creation and deployment:

- **`mcpgen` (CLI utility)**—A new utility designed for the containerized nature of the MCP Server, enabling administrative tasks, such as creating and configuring runtime instances.
  - *Role*: Serves as the profile assembler and lifecycle driver. It creates `generated/<profile-_name>` directories containing configuration files, certificates, and scripts and can also export production-ready bundles. For detailed information about the core CLI command set, refer to Use the mcpgen CLI for profile generation and management on page 119.
  - *Typical action*: Executes commands such as `up`, `list`, `logs`, `export`, `remint-all`, and client test calls.
  - *Runs as*: Local Python using the `stdlib` wheel through a wrapper script (`.bat`/`.ps1`, `.sh`)
- **MCP Server (runtime)**—The MCP Server Docker image that delivers the core functionality of the OpenEdge MCP Server. It supports converting API specifications into MCP tools, managing authentication and authorization, tool scoping, routing requests, and calling downstream APIs.
  - *Role*: Serves generated MCP tools from your OpenAPI specification, including authentication, scopes, and guardrails.
  - *Typical action*: Handles client tool discovery and invocation.
  - *Runs as*: `openedge-mcp-server` container image
- **Utilities image**—A Docker image that assists `mcpgen` to create profiles, generate certificates, create service account JWT tokens, and make MCP Client calls.
  - *Role*: Provides an internal helper container invoked by `mcpgen` for operational tasks such as service account key generation, token/JWT creation, and certificate handling. Not intended for direct standalone use
  - *Typical action*: Performs key and credential bootstrap and diagnostics, which are triggered automatically
  - *Runs as*: `openedge-mcp-utils` container image

## Rationale for component separation

The components are separated to ensure security, efficiency, and streamlined onboarding for different deployment stages. The following points explain why each component has its own role:

- **Lean production image**—The server ships only request-path code for efficiency.
- **Secure separation**—Credential and key minting occurs in a dedicated utilities image, kept out of the runtime environment, and orchestrated by `mcpgen`.
- **Fast onboarding**—`mcpgen` composes all elements deterministically without requiring heavy dependency installations.

---

**Note:** Due to the powerful and administrative nature of `mcpgen`, it is ***not*** recommended to deploy this utility in any production environment running an MCP Server. The export process exists to prepare an MCP Server runtime instance for self-contained operation, eliminating the need for additional tooling.

---

### How the components work together

Understanding how the components interact helps you visualize the overall workflow. This model shows the sequence of actions from development to production:

1. Use the `mcpgen` CLI to generate a profile, manage configuration, and orchestrate lifecycle operations for OpenEdge MCP environments. During profile generation process, `mcpgen` launches the MCP Server and automatically invokes the utilities image whenever privileged operations, such as key generation or certificate handling are required.
2. Protect the execution of the MCP utilities. All operations involving credentials, certificates, or diagnostics are wrapped by `mcpgen` commands for security and simplicity.
3. In production, only the MCP Server container runs continuously to serve MCP tools. The `mcpgen` CLI and utilities image are used occasionally for maintenance tasks such as key rotation or configuration updates, typically from a controlled workstation or CI/CD pipeline.

## Supported environments

The OpenEdge MCP Server supports multiple environments for development and production. Review the following details before installation.

### Development platforms

The OpenEdge MCP Server supports development on the following platforms:

- Linux
- Windows

Progress recommends Linux-based container runtimes for all production deployments. Windows environments may be used for development and testing.

---

**Note:** Production deployments are not supported on Windows because the actual runtime of the MCP server is always on Linux.

---

### Production platforms

The OpenEdge MCP Server container images are OCI-compatible and supported for production deployments on Linux-based OCI-compatible container runtimes. Production deployments require a Linux-based runtime. The container images are compatible with OCI-compliant container engines as well as Kubernetes orchestration platforms.

The following details summarize the deployment requirements and platform considerations for the OpenEdge MCP Server container images:

- **Container image requirements**: The OpenEdge MCP Server container images are Linux-based and require a Linux runtime environment for production deployments.
- **Production deployment best practice**: Production deployments must run on Linux-based container runtimes.

## Prerequisites

The following table lists the required components for installation and their purpose:

| Component | Minimum version | Purpose | Notes |
|---|---|---|---|
| Docker Engine | 24+ (tested up to 26.x) | Runs MCP Server and utilities containers | Required for all security operations (TLS, Service Account tokens) |
| Docker Compose | v2 plugin (preferred) or docker-compose v1 | Orchestrates the profile stack | Detected by the `mcpgen doctor` command |
| Python | 3.11 and later | Executes `mcpgen` and utilities locally | Required; versions outside this range are not supported |
| OpenAPI specification | OpenAPI 3.x with stable, unique `operationId`values | Serves as the source of truth for tool generation | • The OpenAPI specification should be valid and linted.<br>• The `operationID` must be stable and unique across builds. Changing the `operationId` without intent to regenerate tooling is considered a breaking change and may result in invalidated tool mappings, incorrect LLM behavior, or inconsistent MCP Server operation.<br>• The OpenAPI specification is user-supplied and not installed. |
| unzip | Any | Extracts the distribution archive | Not needed if cloning from source |
| Bash shell | Any recent version | Required for running shell scripts on Linux | Should be installed on Linux systems |

### Notes

- As a prerequisite, the environment must pass checks by running `./mcpgen doctor` before first use. For more information, see Step 3: Verify your environment on page 47.
- All TLS certificate generation and Service Account operations use containerized security utilities (`openedge-mcp-utils`). No local OpenSSL installation is required.
- Host Python is mandatory because the `mcpgen` shell wrapper executes Python code from the distributed wheel or source. Supported range is 3.11 and later.
- Windows users can use either Command Prompt or PowerShell. Linux users can use any preferred shell.

## Python configuration

The `mcpgen` CLI requires a supported Python interpreter to run. This section explains how to configure Python for OpenEdge MCP Server.

### Initial configuration

On the first run, `mcpgen` prompts you to specify the Python interpreter. Supported versions are Python 3.11 and later.

```
Enter the Python command or path you want to use.
Examples: python3.12, python3.13, /usr/bin/python3.11

Python command: _
```

Enter the full path to the executable, for example `/usr/bin/python3.11`. Your selection is saved in the `./mcpgen_python` file located in the `mcpgen` installation directory. This configuration is reused for all future runs.

### Reconfigure python

To change the Python interpreter after initial setup, use the following commands:

| Platform | Command |
|---|---|
| Unix/Linux | `./mcpgen --select-python` |
| Windows PowerShell | `./mcpgen.ps1 -SelectPython` |

### Temporary override

To override the Python interpreter for a single command:

| Platform | Command |
|---|---|
| Unix/Linux/macOS | `MCPGEN_PYTHON=/usr/local/bin/python3.12 ./mcpgen list` |
| Windows PowerShell | `$env:MCPGEN_PYTHON="C:\Python311\python.exe"; ./mcpgen.ps1 list` |

### Note

The Python configuration is stored in the `mcpgen` installation directory, not in the current working directory. This ensures consistent behavior regardless of where you run `mcpgen`.

# Install OpenEdge MCP Server

This topic outlines the process for setting up the OpenEdge MCP Server in development environments. The following steps represent high-level actions in the workflow. These steps ensure your MCP Server is properly configured in the development environment, before you enforce security best practices and prepare for deployment to production. For detailed instructions and configuration options, refer to the respective linked topics provided in each step:

1. Step 1: Download and unpack the distribution package on page 44.
2. Step 2: Run the container image load script on page 45.
3. Step 3: Verify your environment on page 47.
4. Step 4: Provide OpenAPI specification on page 48.
5. Step 5: Generate and start a profile on page 49.
6. Step 6: Validate the tool catalog on page 55.

**What's next:** After completing the installation and initial setup in your development environment, start the MCP Server with the generated profile and test it by sending a natural language request mapped to API operations. To do this, use a client that implements the Model Context Protocol (MCP) and has access to large language models (LLMs). For example, Visual Studio Code (VSCode) provides integrated MCP Client support and enables interaction with LLMs for natural language query processing to MCP tools.

Further, it is essential to enforce security best practices and harden your profile before deploying to production. This includes enabling TLS or mTLS, configuring authentication and authorization, rotating keys and tokens, vaulting secrets, and applying runtime guardrails. For more information on the security and pre-production hardening checklist, see Enforce security best practices before deployment on page 57.

Only after you have followed each step in the checklist should you export your hardened profile and proceed with production deployment. Following these measures help ensure your OpenEdge MCP Server is secure, compliant, and production-ready. For more information on deploying OpenEdge MCP Server to production, see Deploy OpenEdge MCP Server to production on page 59.

## Step 1: Download and unpack the distribution package

To install the OpenEdge MCP Server, you must first download and unpack the distribution package. This section explains the available distribution formats and how to prepare the files for use.

### Download the release artifact

Download the release ZIP file from the Progress ESD website and place it in your working directory.

1. Navigate to the Progress Software ESD website.
2. From the list of available product lines, select **Progress OpenEdge > Progress OpenEdge <*version*>**.
3. From the list of releases, select **Progress OpenEdge MCP Server**.
4. From the list of available files, select the appropriate distribution package to download.

### Unpack the distribution

Run the following commands to extract the files.

- On Linux:

```
unzip OPENEDGE_MCP_SERVER_<version>_LNX64.zip -d oemcp-dist
cd oemcp-dist
```

- On Windows (PowerShell):

```
Expand-Archive -Path OPENEDGE_MCP_SERVER_<version>_WIN64.zip -DestinationPath
oemcp-dist
cd oemcp-dist
```

  Alternatively, you can use an unzip utility or select the **Extract All** from the Windows Explorer context menu.

### Contents of the distribution

After unpacking, the directory contains:

| File/Directory | Description |
|---|---|
| `mcpgen\|bat\|ps1` | CLI wrapper for profile generation and lifecycle tasks |
| `load_images.sh\|bat\|ps1` | Script to load pre-built Docker images (server and utilities) |
| `images/*.tar.gz` | Docker images for offline installation |
| `tools/` | Templates and helper libraries |
| `VERSION` | Product version information |
| `mcp.properties.README` | Additional configuration details |
| `CHECKSUM.txt` | Contains the SHA256 message digests for all files in the distribution package. |
| `IMAGES_MANIFEST.txt` | Lists the container images included in the distribution package, if any. |
| `wheels/` | Includes the `mcpgen` Python wheel (`.whl`) for tooling. |
| `examples/` | Provides various examples for using the OpenEdge MCP Server. |

## Step 2: Run the container image load script

After unpacking the distribution, you must load the pre-built Docker images for the MCP Server and utilities. This step ensures that the correct server and utility images are available for deployment. This section explains how to perform the load and verify the installation.

## Prerequisites

Before loading container images, installation of a containerization manager is required where the OpenEdge MCP Server runs. All subsequent instructions assume you are using Docker through Docker Engine or Docker Desktop.

- Refer to the official Docker documentation for installation instructions for your operating system.
- For Windows, you may need to install Windows Subsystem for Linux (WSL 2). For more information, refer to the official WSL installation guide provided by Microsoft.

## Run the script

After you have ensured that the required prerequisites have been met, run the following script to load the container images into your local Docker environment.

**Note:** Ensure that your Docker server is running before running the script.

- On Linux:

```
./load_images.sh
```

- On Windows:

```
load_images.bat
```

This script imports the MCP server and utility images into your local Docker environment.

## Step 3: Verify your environment

Use the following commands to confirm that the images and versions are correctly installed:

---

**Note:** The `mcpgen` utility version number, server image ID, and creation timestamp shown in the examples may differ from those in your environment. Comparing this output with your local setup may show different values.

---

1. Use the built-in `./mcpgen doctor` command to run environment and dependency diagnostics. For more information, see Doctor on page 136.

   ---

   **Note:** Before running any `mcpgen` commands, ensure that Python 3.11 or later is installed and available in your system PATH. For installation and configuration details, see Supported environments on page 41 and Python configuration on page 43.

   ---

2. List the available images in Docker to confirm the MCP Server image and utilities image exist:

   ```
   docker images --filter "reference=openedge-mcp*"
   ```

   An output similar to the following appears:

   ```
   docker images --filter "reference=openedge-mcp*"

   IMAGE                          ID            DISK USAGE   CONTENT SIZE   EXTRA
   openedge-mcp-server:<version> 11f02a0b5f1f  397MB         189MB
   openedge-mcp-server:latest    11f02a0b5f1f  397MB        189MB
   openedge-mcp-utils:<version>  546c3a64f666  481MB         227MB
   openedge-mcp-utils:latest     546c3a64f666  481MB        227MB
   ```

   Where, <*version*> matches the distribution package you downloaded.

3. Verify the `mcpgen` utility version and timestamp of the server image:

   ```
   ./mcpgen --version
   ```

   An output similar to the following appears:

   ```
   mcpgen Python edition: <version>
   Server image: openedge-mcp-server:latest (created: 2025-12-17T18:09:08.986464865Z)
   ```

   Where, <*version*> matches the distribution package you downloaded.

# Step 4: Provide OpenAPI specification

To expose MCP tools, you must supply a valid OpenAPI specification. OpenAPI specifications can be obtained from a Progress Application Server (PAS) for OpenEdge instance, which may implement one or more of the following sources of service information:

- See "Develop a Progress Data Object Service" in *Develop ABL Services*
- See "OpenAPI documentation for REST-based services" in *Manage Progress Application Server (PAS) for OpenEdge*
- See "OpenEdge.Web.DataObject.Writer.OpenAPI30ServiceWriter" in *OpenEdge ABL API Reference*
- See "Use a WebHandler" and "ABL Web App project type" in *Develop ABL Services*
- Progress OpenEdge MCP Server How-To Guide

This section focuses on where to place the OpenAPI data and how to generate a compatible `openapi.yml` file from a catalog for a Progress Data Object Service. You can use Progress Developer Studio for OpenEdge to build an ABL Web App with a Data Object Service and generate the catalog. The catalog can be stored locally or accessed through a URL from a deployed service.

## Place the specification

Place your OpenAPI specification file, for example, `openapi.yml` in the distribution directory or an adjacent directory.

**Minimum requirement:** The specification must include valid `paths:` and `operationId` entries.

## Generate specification from Progress Data Object Service catalog

You can generate an OpenAPI specification for MCP tools using either a local catalog file or a remote catalog endpoint. Use the built-in `openapi-gen` utility to convert your Progress Data Object Service catalog into OpenAPI format. This process creates an `openapi.yml` file that describes the available tools and their operations, which can be used for client integration or API documentation.

Use the following command to generate an OpenAPI specification from a local catalog file:

```
./mcpgen openapi-gen <source> -o <openapi.yml>
```

where,

| Parameter | Description |
|---|---|
| *<source>* | Progress Data Object Service catalog source, which can be either a local file path, such as catalog.json, or an HTTP or HTTPS URL<br><br>**Note:** When using a local catalog file as the source, provide an absolute file path if you are running the command from a directory other than the catalog location. |
| `-o` *<openapi.yml>* | (Optional) Specifies the output file for the generated OpenAPI specification.<br><br>If you do not provide `-o`, the command uses a default, auto-generated filename for the output. |

You can use Progress Developer Studio for OpenEdge to build an ABL Web App with a Data Object Service and generate a catalog. Then provide the URL where that catalog is accessible. The URL to access the catalog should look like this:

```
http://host:port/<web-app>/pdo/<ServiceName>
```

For example, if the `Sports2020` web app with the `Sports2020Service` Data Object Service is deployed and available, you can generate the specification file using:

```
./mcpgen openapi-gen http://localhost:8810/Sports2020/web/pdo/Sports2020Service -o
<openapi.yml>
```

where,

| Parameter | Description |
|---|---|
| `http://localhost:8810/Sports2020/web/pdo/Sports2020Service` | The URL of the remote Progress Data Object Service catalog endpoint. |
| `-o <openapi.yml>` | (Optional) Specifies the output file for the generated OpenAPI specification.<br>If you do not provide `-o`, the command uses a default, auto-generated filename for the output. |

These commands generate a complete OpenAPI 3.0.3 specification with proper schemas, paths, and operations based on your Progress Data Object Service catalog. For more information, see Generate OpenAPI specification on page 129.

For detailed instructions on structuring input and output schemas, assigning titles, writing descriptions, and defining error models, see OpenAPI specification quality guidelines on page 267.

## Step 5: Generate and start a profile

This section explains how to create and start an MCP profile using the `mcpgen` utility. A profile defines the configuration, certificates, and runtime environment for MCP services. Generating a profile is an important step in preparing the MCP server for development or production testing.

### Basic profile setup

Use this command to quickly start the MCP Server using your OpenAPI specification file:

```
./mcpgen up --spec <openapi.yml>
```

where, `--spec ./<openapi.yml>`—Specifies the OpenAPI specification file for tool definitions.

## Basic TLS setup

Use this command to start the MCP Server with TLS enabled and your OpenAPI specification:

```
./mcpgen up --quick --tls --spec ./<openapi.yml>
```

where,

- `--quick`—Creates a profile using default settings for faster setup.
- `--tls`—Enables TLS for secure communication.
- `--spec ./<openapi.yml>`—Specifies the OpenAPI specification file for tool definitions.

## TLS with custom hostnames

Run this command to enable TLS and include additional hostnames in the certificate to avoid browser or client warnings:

```
./mcpgen up --quick --tls --extra-hostnames "api.example.com,myapp.local" --spec
./<openapi.yml>
```

where,

- `--extra-hostnames`—Adds additional hostnames to the TLS certificate Subject Alternative Names (SAN).
- `"api.example.com,myapp.local"`—Comma-separated list of hostnames to include.

## What happens during profile creation

The following steps outline the process of profile creation:

1. A directory named `generated/<profile_name>/` is created. The system also updates the `generated/servers.registry.tsv` file to track the newly created profile.
2. Self-signed TLS certificates are generated for development if none exist.
3. Service account keys and JWT tokens are created (unless `--no-sa` is specified).
4. The `mcp_server_config.json` MCP server configuration file is written with default routing settings.
5. The container starts and performs a smoke check, where:
   - All the tools available on the MCP Server are listed.
   - The client invokes the first MCP tool, which results in an API call to the MCP Server. The server processes the request and returns the response to the client.

---

**Note:** If you used `localhost` in the base URL when generating the profile, the MCP Server container cannot reach the Data Service. For more information, see Work with a data service created using localhost on page 53.

---

## Other configuration options and generated profile structure

During profile creation, additional configuration options are applied. The following table provides details of the directory structure and client MCP configuration created by the `up` command.

| Configuration | Purpose |
|---|---|
| Service account keys and tokens | Used for MCP client-to-server authentication |
| TLS/mTLS settings | Configures transport security for the MCP server |
| OpenAPI visibility and context settings | Ensures proper exposure and context handling for API operations |
| Downstream TLS certificates and host verification | Provides secure communication with downstream services |

The `up` command creates a structured directory under `generated/<profile>/`. This directory contains all necessary artifacts, such as configuration files, scripts, certificates, and metadata required for consistent deployment and operation. The following table lists the contents of the generated profile:

| Directory/File | Description |
|---|---|
| `sa/` | Contains key material if the service account feature is enabled. |
| `client/` | Contains configuration and environment variables for client-side operations, enabling interaction with the MCP Server without manual setup. |
| `server/` | Contains configuration files, environment variables, and scripts required to start and manage the MCP Server instance. |
| `certs/` | Contains TLS and mTLS assets if secure communication is selected. |
| `start_server.sh\|ps1` | Script to start the MCP Server instance. |
| `try_client.sh\|ps1` | Script to validate client-server interaction. |
| `docker-compose.mcp.yml` | Docker configuration to run the OpenEdge MCP server with required parameters and environment details. |
| `profile.json` | Contains metadata file describing the generated profile. |

**Contents of the `server` directory**

The `server` directory holds all files required to start and configure the MCP Server:

| File | Description |
|---|---|
| *<openapi.yml>* | The OpenAPI specification used to define and generate the MCP Server. |
| `server.env` | Stores environment variables required for the MCP server, including sensitive information such as tokens, certificates, and other secrets. |
| `mcp_server_config.json` | Contains the core MCP Server configuration, which is used to create and start the MCP Server. The MCP Server cannot run without this file. Use the `.verbose` option for detailed output. |

**Contents of the `client` directory**

The `client` directory provides everything needed to interact with the MCP Server without manual setup:

| File | Description |
|---|---|
| `sa.jwt` | Stores a JWT associated with the service account configured for the MCP Server. This token is required for secure communication between the client and the server. |
| `client.env` | Stores environment variables for the MCP client. |
| `mcp.json` | Auto-generated MCP client configuration file. |

The `mcp.json` file is an auto-generated client configuration that simplifies connecting an MCP client to your development environment. For example, in VS Code, you can use this file to configure the MCP client without manually setting up endpoints, authentication tokens, or tool definitions.

Instead of creating these settings from scratch, the MCP generation process provides a ready-to-use `mcp.json`. You can simply copy this file into the MCP client configuration of your IDE, and start interacting with the MCP Server immediately without any additional setup.

The following characteristics define how authentication headers are managed in the `mcp.json` client MCP configuration file:

- The service account header is included only when service account authentication is enabled.
- Authentication header precedence is resolved in the following order: *`service_account.prefix`* → *`user_auth.prefix`* → `Bearer`
- Double prefixing is automatically avoided.

For example:

```
{
  "servers": {
    "myprof": {
      "url": "http://localhost:8500/http",
      "type": "http",
      "headers": {
        "X-OEMCP-SERVICEACCOUNT": "Bearer eyJ..."
      }
    }
  },
  "inputs": []
}
```

## List all profiles

To list all profiles, run:

```
./mcpgen list --wide
```

For detailed information about the core commands used to manage a profile with the `mcpgen` CLI, see Manage core commands in mcpgen on page 120.

## Work with a data service created using localhost

When running the MCP Server in a Docker container, the container cannot reach services running on the host machine using localhost in the base URL. Each container runs in an isolated network, so `localhost` always resolves to the container itself, not the host machine.

This becomes an issue only if the MCP Server profile was originally created using a base URL that contains `localhost`. For example, during development, an OpenEdge Data Service might be running directly on the host and used to generate the MCP server profile:

```
./mcpgen openapi-gen http://localhost:8810/Sports/web/pdo/SportsService
./mcpgen up --spec ./openapi_SportsService.yml
```

Although `localhost` works at profile creation time, once the MCP Server starts inside a container, MCP tools running in that container can no longer reach the Data Service using `localhost`.

If the MCP Server profile was originally created using a host name or an IP address, no additional configuration is required.

To support development scenarios where `localhost` has been used, MCP provides a convenience mechanism using a mapped host name (`data.service`) using the `extra_hosts` configuration.

## Recommended approach: Use a host name or IP address

To access the Data Service from the MCP Server container, update the base URL to use a host name or IP address that points to the host machine.

**Example**

```
./mcpgen update --base-url http://192.168.10.100:8810/Sports/web/pdo/SportsService
openedge-mcp-server
```

If the MCP Server profile was created using a host name or IP address from the beginning, no additional configuration is required.

## Development-only approach for localhost base URLs

During development, as an alternative to changing the base URL to a host name or IP address, you can use the special host name `data.service`. This name is mapped to the host machine using the `extra_hosts` setting in the Docker Compose file.

Perform the following steps:

1. Edit the `docker-compose.mcp.yml` Docker Compose file for the profile located at:

```
generated/<profile>/docker-compose.mcp.yml
```

2. Uncomment the extra_hosts block and update the data.service IP address to either the Docker host gateway IP (for example, `172.18.0.1`) or the IP address of the host:

```
extra_hosts:
 - "data.service:172.18.0.1"
```

3. Update the base URL to use `data.service`:

```
./mcpgen update --base-url <URL> <profile>
```

   For example,

```
./mcpgen update --base-url http://data.service:8810/Sports/web/pdo/SportsService
openedge-mcp-server
```

4. Restart the OpenEdge MCP Server container:

```
./mcpgen restart –no-smoke <container-name>
```

   For example,

```
./mcpgen restart -no-smoke openedge-mcp-server
```

5. After the server restarts, use the MCP client to call a tool and confirm that the data service is accessible:

```
. /mcpgen client --tool <tool-name> --json <profile>
```

   For example,

```
./mcpgen client --tool getCustomer --json openedge-mcp-server
```

### Production behavior

The `extra_hosts` configuration is intended for *development only*. When exporting a profile for production, the `extra_hosts` block is removed automatically. No manual cleanup is required.

## Step 6: Validate the tool catalog

Testing the tool catalog ensures that the MCP client can list available tools and invoke them correctly. This step validates authentication and tool functionality before moving to production.

## List tool names

Run this command to display all available tools for the specified profile:

```
./mcpgen client <profile-name>
```

## Invoke tool

Use this command to invoke a specific tool and pass arguments. The following example calls the `listProducts` tool with a limit of two items:

```
./mcpgen client <profile-name> --tool listProducts --args '{"limit":2}'
```

Where,

- `--tool listProducts`—Specifies the tool to invoke.
- `--args '{"limit":2}'`—Provides arguments in JSON format for the tool.

## Use different service account tokens

Service account tokens control access to MCP tools based on assigned scopes. If you have multiple tokens with different permissions, you can specify which token to use when listing tools or invoking them. This step allows you to test access levels and validate role-based functionality. If multiple service account tokens exist with different scopes, select a specific token using `--sa-token` option with the `mcpgen client` command:

| Action | Command |
|---|---|
| List tools with regular access | `./mcpgen client <profile_name> --sa-token regular` |
| List tools with admin access | `./mcpgen client <profile_name> --sa-token admin` |
| Invoke admin-only tool | `./mcpgen client <profile_name> --sa-token admin --tool getSupportTickets --args '{"limit":5}'` |

## List available tokens for a profile

To view all service account tokens associated with a profile, use the following command:

```
./mcpgen sa-tokens <profile_name>
```

For more information, see Client on page 137.

# Enforce security best practices before deployment

After you have set up and validated the OpenEdge MCP Server in your development environment, the next step is to secure and harden your deployment before moving to production. Securing your OpenEdge MCP Server is essential to protect sensitive data, ensure compliance, and maintain operational integrity. This section outlines the critical security controls you must enforce before deploying to production. Follow each step in the checklist to harden your environment and meet organizational and industry standards. For detailed steps and configuration examples, refer to the references provided.

| Step | Action | Purpose | References |
|---|---|---|---|
| 1 | Enable TLS or mTLS | Enabling TLS or mTLS ensures that all network traffic is encrypted and that both client and server identities are verified. Production environments must never allow plain text communication. | • Configure TLS and mTLS certificates on page 182<br>• TLS and mTLS for secure communication on page 208<br>• TLS and mTLS certificate rotation on page 260 |
| 2 | Configure service account authentication | Configuring service account authentication restricts access to authorized clients by requiring JWT tokens for every request. | • The security group on page 164<br>• Security and authentication architecture for MCP server on page 203 |
| 3 | Define and enforce scopes | Defining and enforcing scopes limits access to tools based on user roles and functions, following the principle of least privilege. | • Authorization on page 181<br>• Scope gating for discovery and execution on page 206 |
| 4 | Rotate keys and tokens regularly | Rotating keys and tokens on a regular schedule helps prevent credential misuse and supports compliance with security policies. | • Upgrade process and key or certificate rotation on page 257 |
| 5 | Vault secrets securely | Vaulting secrets ensures that private keys and tokens are protected from unauthorized access and accidental leakage. | • Export and harden production deployment on page 231 |
| 6 | Harden runtime guardrails | Hardening runtime guardrails limits the size of responses and disables unsafe operations to reduce the risk of abuse. | • Configure response guard settings on page 189<br>• Response guard strategy on page 209 |
| 7 | Restrict write routes | Restricting write routes minimizes risk by disabling unnecessary write operations in the production environment. | • Configure routes on page 161<br>• Threat model and mitigation strategies on page 204 |

| Step | Action | Purpose | References |
|---|---|---|---|
| 8 | Sanitize logs | Sanitizing logs prevents sensitive data from being exposed in log files by redacting confidential fields and disabling verbose payload logging. | • Configure logging and related middleware on page 91<br>• Audit events and sensitive data handling on page 93 |
| 9 | Pin Docker image digests | Pinning Docker image digests locks your deployment to known, secure versions and prevents supply chain tampering. | • Export and harden production deployment on page 231 |
| 10 | Validate authentication headers | Validating authentication headers ensures that all clients use the correct headers for secure access to the MCP Server. | • Authentication headers and access information on page 164<br>• Configure service account authentication on page 165 |
| 11 | Run security checks and smoke tests | Running security checks and smoke tests helps detect misconfigurations and security gaps before deployment. | • Doctor on page 136 |

# Deploy OpenEdge MCP Server to production

After you have secured and hardened your deployment, exporting a validated development profile for production is an important step in real-world deployments, which ensures that the artifacts you ship to an operations environment are secure, immutable, and aligned with compliance requirements. It also removes development-specific components and creates a controlled boundary for signing and scanning. This section explains why and how to export artifacts for production environments after successful validation in development.

### Why export instead of running development profiles in production?

The following reasons explain why you should export a production bundle instead of directly running a development profile in a production environment.

| Reason | Description |
|---|---|
| Freeze image digests | Guarantees immutability and aligns with software bill of materials (SBOM) requirements |
| Strip developer artifacts | Removes client scripts, verbose configs, and private keys unless explicitly included |
| Self-contained package | Produces a directory or tarball ready for shipping to operations |
| Controlled artifact boundary | Enables signing, scanning, and provenance checks |

**Caution:** You are recommended to ***never*** run development profiles in production. Exporting enforces compliance and security best practices.

## Prepare a production-ready export

Before deploying the OpenEdge MCP Server to a production environment, it is important to create a production-ready export of your validated development profile. This process ensures that only secure, hardened, and compliant artifacts are included in your deployment package, removing any development-specific components and sensitive data.

### Export command

Run the following command to prepare production-ready artifacts after validating the profile in development:

```
./mcpgen export <profile_name> --bundle prod
```

For more information about the `export` command, see Export profiles for production on page 127.

### Export output

The export process generates a set of files that form the production-ready bundle. The following table lists the included files:

| File/Directory | Purpose |
|---|---|
| `docker-compose.mcp.yml` | Docker Compose configuration for MCP services |
| `profile.json` | Sanitized metadata for deployment.<br>**Note:** This file is included for only production-ready exports. |
| `server/mcp_server_config.json` | MCP server configuration file |
| `certs/` | Public certificate components |
| `run.sh` | Launcher script for Linux and macOS environments. |
| `run.ps1` | Launcher script for Windows environments |

### Optional flags for secrets

You can use the following flags to include secrets in the export. Use these options cautiously and prefer secure vaulting in production environments:

| Flag | Purpose |
|---|---|
| `--include-private` | Retains private keys |
| `--include-token` | Retains tokens |

**Note:** Including secrets in exported artifacts increases risk and is ***not*** recommended. Always use secure vaulting mechanisms in production.

## Run exported bundle on production host

After exporting a validated profile for production, you must deploy the bundle on the target host. This process is designed to work in air-gapped environments and provides flexible options for starting MCP services using Docker Compose or helper scripts.

Run the following steps to deploy the exported production bundle on the target host:

1. Transfer the exported directory to the target host. Ensure that Docker and Docker Compose are installed and configured on the host system.
2. Navigate to the exported bundle directory:

```
cd exported/<name>
```

3. Choose either of the following methods to start the MCP server:
   - **Option A:** Docker Compose (Preferred)

     Run the following command to start the containers in detached mode:

```
docker compose -f docker-compose.mcp.yml up -d
```

   - **Option B:** Helper Script with optional overrides

     The helper script provides a simplified way to start and manage containers. For example,

     On Linux or macOS:

```
./run.sh start
./run.sh start --port 9000 --name my-server
./run.sh status   # Check container status
```

     On Windows using PowerShell:

```
.\run.ps1 start
.\run.ps1 start -port 9000 -name my-server
.\run.ps1 status   # Check container status
```

   where,

   - The first command starts the MCP services using the default configuration from the exported bundle. It launches the Docker containers without any custom overrides.
   - The second command starts the MCP services with custom settings. The default port is 8500. The `-port 9000` option overrides the default and sets the port to `9000`, while the `-name my-server` option assigns a custom name to the server instance. Use this command for running multiple instances or avoiding port conflicts.
   - The third command checks the current status of the MCP containers.

---

**Note:** Option **A** is the preferred method for production deployments because it provides better control and visibility.

---

4. Monitor logs to verify successful startup of the containers.

```
.\run.ps1 start docker compose -f docker-compose.mcp.yml logs -f --tail=50 || \
docker-compose -f docker-compose.mcp.yml logs -f --tail=50
```

Both commands are used to display and follow the logs of containers defined in the `docker-compose.mcp.yml` file. The first command uses the modern Docker Compose v2 syntax (`docker compose`), and the second uses the legacy Docker Compose CLI (`docker-compose`). They are combined with `||` so that if the first command fails, the second one runs as a fallback. Both commands show the last 50 lines of logs and continue streaming new log entries in real time.

# Reconfigure MCP Server without profile regeneration

You can update the MCP Server configuration without recreating the profile. This step is useful for making changes such as adding scopes, adjusting routing, or enabling mTLS after initial deployment. The process differs slightly depending on whether you are working with development profiles or exported production bundles.

## Update configuration for development profiles

Use the `mcpgen` utility to stop the running profile, edit the configuration file, and restart the profile.

1. Stop the running profile.

```
./mcpgen stop <profile_name>
```

2. Edit the `server/mcp_server_config.json` file to apply changes such as adding scopes, adjusting routing rules, or enabling mTLS.
3. Restart the profile with the updated configuration.

```
./mcpgen start <profile_name>
```

### Update configuration for exported bundles

For production bundles, use the launcher scripts provided in the export directory.

1. Stop the running containers using the appropriate helper script.

   On Linux or macOS:

   ```
   ./run.sh stop
   ```

   On Windows PowerShell:

   ```
   .\run.ps1 stop
   ```

2. Edit the `server/mcp_server_config.json` file to apply configuration changes.
3. Restart the containers using the same helper script.

   On Linux or macOS:

   ```
   ./run.sh start
   ```

   On Windows PowerShell:

   ```
   .\run.ps1 start
   ```

### Notes

- Always validate the syntax of `mcp_server_config.json` after editing to avoid startup errors.
- Restarting the profile or containers is required for changes to take effect.

## Version verification and troubleshooting

This topic explains how to verify component versions and resolve version-related issues after installation.

### Verify docker image versions

Run the following command to list all OpenEdge MCP images with their versions:

```
docker images --format "table {{.Repository}}:{{.Tag}}\t{{.ID}}
\t{{.Size}}\t{{.CreatedAt}}" | grep openedge-mcp
```

The following output appears:

```
# openedge-mcp-server:1.0.0    <image-id>  <size>  <date>
# openedge-mcp-server:latest   <image-id>  <size>  <date>
# openedge-mcp-utils:1.0.0     <image-id>  <size>  <date>
# openedge-mcp-utils:latest    <image-id>  <size>  <date>
```

## Verify components versions

Use these commands to check versions of different components:

- To check distribution version, run:

```
cat VERSION
```

- To check version of the `mcpgen` utility, run:

```
./mcpgen --version
```

- To check version of the running server while the container is active, run:

```
docker run --rm openedge-mcp-server:latest python -c "
import sys, os
sys.path.insert(0, '/app/src')
try:
    with open('/app/VERSION', 'r') as f:
        print(f'Server Version: {f.read().strip()}')
except FileNotFoundError:
    print('Version file not found in container')
"
```

## Profile and configuration version tracking

Use these commands to track profile creation details and verify configuration versions:

| Action | Command |
|---|---|
| List profiles | `./mcpgen list --wide` |
| Check configuration version | `cat generated/<profile-name>/mcp_server_config.json \| grep -A5 -B5 version \|\|echo "No version field in config"` |

## Troubleshoot version mismatches

If you encounter version-related issues, follow these steps:

1. If you encounter mixed versions across components, ensure all components use the same release:

```
# Clean up old images
docker images | grep openedge-mcp | grep -v "v0.0.2\|latest" |
awk '{print $3}' | xargs -r docker rmi
```

2. If version tags are missing from your images, re-run the image loading process:

```
./load_images.sh
```

3. If there are conflicts with the mcpgen utility, check if the correct wheel is installed:

```
pip show mcpgen 2>/dev/null || echo "mcpgen not installed via pip"
```

**Note:** The `VERSION` file in your distribution is the single source of truth. All Docker images, `mcpgen` utility, and documentation should align with this version.

III

# Upgrade OpenEdge MCP Server

For details, see the following topics:

- Upgrade OpenEdge MCP Server
- Post-upgrade best practices
- Troubleshoot OpenEdge MCP Server upgrade

# Upgrade OpenEdge MCP Server

This section provides procedures for upgrading OpenEdge MCP Server deployments while preserving existing profiles and supporting verified rollback. An unchanged profile configuration does not normally need to be regenerated during an upgrade.

## Overview

This content covers upgrade procedures for both Linux and Windows PowerShell environments. The following upgrade scenarios are addressed:

1. In-place upgrade of existing standard profiles
2. Profiles using prompts and context enrichment
3. Production bundles created with `mcpgen export --bundle prod`
4. Rollback to the previous working deployment

### Prerequisites

Before starting the upgrade process, ensure you have met all the necessary requirements:

- **Linux system requirements**: Ensure Docker, `unzip`, and Python 3 are installed.
- **Windows system requirements**: Ensure Docker Desktop or Rancher Desktop, and Python 3.11 or later are installed.
- **Schedule a maintenance window** for the container restart.
- **Obtain the new distribution** zip file appropriate for your host and server image.
- **Retain the previous distribution** zip and Docker image until rollback validation is complete.
- **Document your deployment**: Record the profile name, host port, OpenAPI source, at least one known read-only tool, and valid tool arguments for testing.

## Linux upgrade procedures

This section provides step-by-step upgrade procedures for Linux-based OpenEdge MCP deployments. Before you begin, set the deployment variables:

```
export OEMCP_HOME=~/oemcp-dist
export PROFILE=my-oemcp-profile
export TOOL='<known-read-only-tool>'
export TOOL_ARGS='<valid-json-arguments>'

# Example for the Sports test data service:
# export TOOL=getCustomer
# export TOOL_ARGS='{"filter":"CustNum=3000"}'
```

For detailed instructions, refer to the subsequent steps.

### Step 1: Back up the current deployment

Creating a complete backup before the upgrade ensures you can restore your deployment if validation fails. Follow these steps to backup before stopping the profile and do not store private keys or JWTs in source control:

1. Set the working directory to the OpenEdge MCP Server installation directory:

```
cd "$OEMCP_HOME"
```

2. Generate a timestamp and archive the generated profile and server registry:

```


# Create backup archive with timestamp
BACKUP_DATE=$(date +%Y%m%d-%H%M%S)
tar -czf "../${PROFILE}-backup-${BACKUP_DATE}.tgz" \
    "generated/$PROFILE" \
    generated/servers.registry.tsv
```

3. Save a known successful tool invocation as functional evidence:

```
# Capture known successful tool invocation as functional baseline
./mcpgen client "$PROFILE" --tool "$TOOL" --args "$TOOL_ARGS" --json \
  > "../${PROFILE}-tool-call-pre-upgrade.json"
```

4. Confirm that all expected backup artifacts were created before you continue:

```
ls -l "../${PROFILE}-backup-${BACKUP_DATE}.tgz" \
      "../${PROFILE}-tool-call-pre-upgrade.json" \
```

The backup should capture:

- Profile server configuration
- Service-account keys and token metadata
- Certificates (public and private)
- Prompt files (if used)
- Context files (if used)
- File ownership and permissions (preserved by tar, subject to the privileges of the user creating and restoring the archive)

## Step 2: Perform in-place upgrade

This step replaces the existing distribution with the new version while preserving your profile configuration. Follow these steps to stop the profile, replace the distribution tooling, and load the new Docker images:

1. Set the working directory to the OpenEdge MCP Server installation directory:

```
cd "$OEMCP_HOME"
```

2. Stop the profile before replacing distribution artifacts:

```
./mcpgen stop "$PROFILE"
```

3. Remove existing `mcpgen` wheel files to avoid stale tooling conflicts:

```
# Remove previous version wheel file
rm -f wheels/mcpgen-*.whl
```

4. Extract the target OpenEdge MCP Server distribution into the installation directory:

```
# Extract new distribution
unzip -o /path/to/OPENEDGE_MCP_SERVER_<VERSION>_LNX64.zip -d "$OEMCP_HOME"
```

5. Load the target Docker images and validate the environment:

```
# Load new Docker images
./load_images.sh

# Verify installation
printf 'python3\n' | ./mcpgen doctor
```

6. Start the profile after the upgrade completes:

```
# Start the upgraded profile
./mcpgen start "$PROFILE"
```

**Note:** By default, generated profiles use `openedge-mcp-server:latest`. The `load_images.sh` script loads the new image under this reference, so no compose-file changes are required for standard in-place upgrades.

## Step 3: Validate standard profile upgrade

Validation confirms that the upgrade was successful and that all functionality remains intact. To verify that the upgrade preserved functionality and configuration, run the same known tool invocation used in baseline validation:

```
# Verify known tool invocation still works
./mcpgen client "$PROFILE" --tool "$TOOL" --args "$TOOL_ARGS" --json \
  > "../${PROFILE}-tool-call-post-upgrade.json"
```

### Validation checklist

To confirm the upgrade completed successfully, verify the following:

- Verify the known tool invocation succeeds with expected results.
- Confirm TLS/client connectivity is functional.
- Test that invalid service-account tokens are properly rejected.
- Verify profile configuration, certificates, SA key material, and tokens remain unchanged (apart from intentional configuration updates).

Version 1.1.0 changes the default `request_timeout` from 500 ms to 30,000 ms (30 seconds). Profiles that do not explicitly configure this value will observe significantly longer timeout behavior. To retain the previous 500 ms timeout, add the following to your profile's `generated/$PROFILE/server/mcp_server_config.json`:

```
{
  "runtime": {
    "middleware": {
      "request_timeout": {
        "enabled": true,
        "timeout_ms": 500
      }
    }
  }
}
```

For more information, see Configure middleware settings on page 190.

**Important:**

- The chosen timeout must account for downstream OpenEdge service response times and client timeout expectations. The new 30-second default accommodates longer-running operations but may not be appropriate for all deployment scenarios.
- A successful tool call alone does not prove that authorization is enforced. Always verify that invalid credentials are rejected.

## Step 4: Validate profiles with prompts and context enrichment

Profiles using prompts and context enrichment require additional validation to ensure these features continue working correctly. For profiles created with `--prompts <directory>` and `--context <directory>`, perform the standard upgrade procedure from Step 2, then complete the following checks:

1. Confirm files under `server/prompts/` and `server/context/` match the pre-upgrade baseline.
2. Confirm `prompts/list` returns all expected prompts.
3. Confirm `prompts/get` returns the correct content for each expected prompt.
4. Confirm `tools/list` includes expected tools and that context text from `server/context/global.md` appears in enriched tool descriptions.

## Step 5: Upgrade production exports

Production exports require a different upgrade approach because they are self-contained deployment bundles with independent configurations. Follow these steps to upgrade production exports:

1. Set the working directory to the OpenEdge MCP Server installation directory:

```
cd "$OEMCP_HOME"
```

2. Find the timestamped production export directory for the profile:

```
# Find the production export directory
EXPORT_DIR=$(find exports -maxdepth 1 -type d \
  -name "export_${PROFILE}_*" -print -quit)
test -n "$EXPORT_DIR"
```

**Note:** This command selects the first matching export. If you have multiple exports for the same profile, set `EXPORT_DIR` explicitly to the target directory.

3. Stop the running container and back up the export compose file:

```
cd "$EXPORT_DIR"

# Stop the running container
./run.sh stop --port <port> --name <container-name>

# Back up the compose file
cp docker-compose.mcp.yml docker-compose.mcp.yml.backup
```

4. Confirm the export follows security best practices and does not include disallowed private credentials:

```
test ! -e sa/private.pem  # Should not exist
test ! -e sa/sa.jwt       # Should not exist
test -f sa/public.pem     # Should exist
test -f server/mcp_server_config.json  # Should exist
```

Production exports intentionally omit the service-account private key and JWT. These credentials must be provided from an external secret-management system. Never copy development credentials into production bundles.

5. Load the target release images into Docker:

```
# Load new Docker images
cd "$OEMCP_HOME"
./load_images.sh
```

**Note:** This procedure assumes that the OpenEdge MCP Server distribution is available on the production machine.

6. Restart the production export with the upgraded image:

```
# Start the upgraded container
cd "$EXPORT_DIR"
./run.sh start --port <port> --name <container-name>
```

7. Confirm the container image reference and runtime status:

```
# Verify deployment - confirm container uses the new image
CONTAINER_IMAGE_ID=$(docker inspect --format '{{.Image}}' <container-name>)
EXPECTED_IMAGE_ID=$(docker image inspect openedge-mcp-server:latest --format '{{.Id}}')

if [ "$CONTAINER_IMAGE_ID" = "$EXPECTED_IMAGE_ID" ]; then
    echo "Container is using the upgraded image"
else
    echo " WARNING: Container image mismatch"
    echo "  Container: $CONTAINER_IMAGE_ID"
    echo "  Expected:  $EXPECTED_IMAGE_ID"
fi
# Check runtime status
./run.sh status --port <port> --name <container-name>
```

## Production validation checklist:

To confirm the production upgrade completed successfully, verify the following:

- Verify the deployed container uses the new image
- Confirm sanitized server configuration is unchanged
- Test that externally supplied credentials can list tools
- Verify authorized tool invocation succeeds
- Compare production tool catalog with pre-upgrade baseline
- Confirm production networking (DNS, service discovery, network aliases) functions correctly

**Note:** Production exports remove development-only `extra_hosts` mappings. Use stable DNS names, service-discovery names, or supported network aliases for downstream OpenEdge services.

## Step 6: Pin explicit image references (Optional)

Pinning to explicit image versions provides greater control over which version runs in your deployment. Using `openedge-mcp-server:latest` is appropriate when `load_images.sh` is the controlled mechanism for loading releases. However, explicit tags or digests provide:

- Reproducible records of exact images in use
- Prevention of unintended version changes from later image loads
- Support for release-approval processes

Follow these steps to pin an explicit image reference:

1. Set the desired image tag or digest:

```
export SERVER_IMAGE_REFERENCE=openedge-mcp-server:<version>
```

2. Apply the image reference to the generated profile compose file:

```
# Update profile compose file
sed -i -E \
  "s|^([[:space:]]*image:).*|\1 ${SERVER_IMAGE_REFERENCE}|" \
  "generated/$PROFILE/docker-compose.mcp.yml"
```

3. Apply the same image reference to the production export compose file:

```
# Or update production export compose file
sed -i -E \
  "s|^([[:space:]]*image:).*|\1 ${SERVER_IMAGE_REFERENCE}|" \
  "$EXPORT_DIR/docker-compose.mcp.yml"
```

   Always back up the original compose file before modifying image references to support rollback.

4. Recreate the container to apply the image reference change:

   - For standard profiles:

```
cd "$OEMCP_HOME"
./mcpgen stop "$PROFILE"
./mcpgen start "$PROFILE"

# Verify the container uses the pinned image
docker inspect --format '{{.Config.Image}}' "${PROFILE}"
```

   - For production exports:

```
cd "$EXPORT_DIR"
./run.sh stop --port <port> --name <container-name>
./run.sh start --port <port> --name <container-name>

# Verify the container uses the pinned image
docker inspect --format '{{.Config.Image}}' <container-name>
```

**Note:** Editing the compose file does not change already-running containers. The container must be stopped and restarted for the new image reference to take effect.

## Step 7: Rollback procedures

If validation identifies issues with the upgrade, you can restore your previous working deployment using one of the following rollback methods.

### Image-only rollback

Follow these steps to restore the previous Docker image while keeping the existing profile configuration unchanged:

1. Stop the running profile:

```
cd "$OEMCP_HOME"
./mcpgen stop "$PROFILE"
```

2. Reload the previous image:

```
# Reload previous version image
/path/to/previous-distribution/load_images.sh
```

3. Restart the profile:

```
./mcpgen start "$PROFILE"
```

   Verify the running container uses the previous image and repeat the known successful tool invocation.

**Note:** If the profile uses an explicit image reference, either restore the previous compose file or update the image reference to point to the previous version image.

### Complete profile rollback

Follow these steps to restore the profile directory from backup and return the deployment to its previous working state:

1. Stop the running profile:

```
cd "$OEMCP_HOME"
./mcpgen stop "$PROFILE"
```

2. Move the current generated profile aside for analysis:

```
# Preserve the upgraded profile for analysis
mv "generated/$PROFILE" "generated/${PROFILE}.post-upgrade"
```

3. Restart the profile:

```
# Restore from backup
tar -xzf "../${PROFILE}-backup-<timestamp>.tgz"

./mcpgen start "$PROFILE"
```

   Verify the restored profile starts successfully and its tool catalog matches the pre-upgrade baseline.

4. Restore the previous mcpgen wheel files:

```
# Restore wheel files from the previous distribution
cp /path/to/previous-distribution/wheels/mcpgen-*.whl "$OEMCP_HOME/wheels/"
```

**Note:** If you removed the previous distribution, you can restore the mcpgen tooling by reinstalling from the archived distribution zip file.

## Production bundle rollback

Follow these steps to restore production export artifacts and, when needed, the previous container image:

1. Stop the production container:

```
cd "$EXPORT_DIR"
./run.sh stop --port <port> --name <container-name>
```

2. Restore the compose file:

```
# Restore previous compose file
cp docker-compose.mcp.yml.backup docker-compose.mcp.yml
```

3. Reload previous image (if using latest):

```
# If using :latest reference, reload previous image
/path/to/previous-distribution/load_images.sh
```

4. Restart the production container:

```
./run.sh start --port <port> --name <container-name>
```

Verify the production container uses the previous image and can successfully invoke authorized tools.

# Windows upgrade procedures

This section provides step-by-step upgrade procedures for Windows-based OpenEdge MCP deployments using PowerShell. Before you begin, ensure that:

- Docker Desktop must be running
- Python 3.11 or later must be available to `mcpgen.ps1`.
- Set deployment variables:

```
$env:OEMCP_HOME = Join-Path $HOME 'oemcp-dist'
$Profile = 'my-oemcp-profile'
$Tool = '<known-read-only-tool>'
$ToolArgs = '<valid-json-arguments>'

# Example for the Sports test data service:
# $Tool = 'getCustomer'
# $ToolArgs = '{"filter":"CustNum=3000"}'
```

**Note:** If your system does not allow running PowerShell scripts (`.ps1` files), use `.\mcpgen.bat` instead of `.\mcpgen.ps1` throughout these procedures.

For detailed instructions, refer to the subsequent steps.

## Step 1: Back up the current deployment

Creating a complete backup before the upgrade ensures you can restore your deployment if validation fails. Follow these steps to back up the current deployment:

1. Set the working directory to the OpenEdge MCP Server installation directory:

```
Set-Location $env:OEMCP_HOME
```

2. Generate a timestamp and archive the generated profile and server registry:

```
# Create timestamped backup
$BackupDate = Get-Date -Format 'yyyyMMdd-HHmmss'
Compress-Archive -Path @(
    "generated\$Profile",
    'generated\servers.registry.tsv'
) -DestinationPath "..\$Profile-backup-$BackupDate.zip" -Force
```

Protect the backup archive and retain the previous distribution until rollback validation is complete. Ensure private keys and tokens are protected from unauthorized access.

## Step 2: Perform in-place upgrade

Follow these steps to replace the existing distribution with the new version while preserving your profile configuration:

1. Move to the installation directory and stop the running profile:

```
Set-Location $env:OEMCP_HOME
.\mcpgen.ps1 stop $Profile
```

2. Remove existing `mcpgen` wheel files to avoid stale tooling conflicts:

```
# Remove previous version wheel files
Get-ChildItem -Path 'wheels\mcpgen-*.whl' | Remove-Item
```

3. Extract the target OpenEdge MCP Server distribution into the installation directory:

```
# Extract new distribution
Expand-Archive -LiteralPath "$HOME\Downloads\OPENEDGE_MCP_SERVER_<VERSION>_WIN64.zip"
 -DestinationPath $env:OEMCP_HOME -Force
```

4. Load the target Docker images and validate the environment:

```
# Load new Docker images
.\load_images.ps1

# Verify installation
.\mcpgen.ps1 doctor
```

5. Start the profile after the upgrade completes:

```
# Start the upgraded profile
.\mcpgen.ps1 start $Profile
```

   The default compose configuration uses `openedge-mcp-server:latest`. The `load_images.ps1` script loads the new image under this reference, so no compose-file changes are required.

## Step 3: Validate standard profile upgrade

Validation confirms that the upgrade was successful and that all functionality remains intact. To verify that the upgrade preserved functionality and configuration, run the same known tool invocation used in baseline validation:

```
# Verify known tool invocation still works
.\mcpgen.ps1 client $Profile --tool $Tool --args $ToolArgs --json |
  Set-Content -Path "..\$Profile-tool-call-post-upgrade.json" -Encoding utf8
```

### Validation checklist:

To confirm the upgrade completed successfully, verify the following:

- Confirm known tool invocation succeeds
- Verify invalid service-account tokens are rejected
- Confirm profile configuration, certificates, keys, and tokens are consistent with the backup
- Review release notes for any default value changes that may affect behavior

Version 1.1.0 changes the default `request_timeout` from 500 ms to 30,000 ms (30 seconds). Profiles that do not explicitly configure this value will observe significantly longer timeout behavior. To retain the previous 500 ms timeout, add the following to your profile's `generated\$Profile\server\mcp_server_config.json`:

```
{
  "runtime": {
    "middleware": {
      "request_timeout": {
        "enabled": true,
        "timeout_ms": 500
      }
    }
  }
}
```

For more information, see Configure middleware settings on page 190.

**Important:**

- The chosen timeout must account for downstream OpenEdge service response times and client timeout expectations. The new 30-second default accommodates longer-running operations but may not be appropriate for all deployment scenarios.

## Step 4: Validate profiles with prompts and context

For profiles using prompts and context enrichment, perform the standard upgrade from Step 2, then complete the following checks:

1. Confirm files in `server/prompts/` and `server/context/` match the backup.
2. Confirm `prompts/list` returns all expected prompts.
3. Confirm `prompts/get` returns the correct prompt content.
4. Confirm `tools/list` includes expected global-context text in tool descriptions.

## Step 5: Upgrade production exports

Production exports require a different upgrade approach because they are self-contained deployment bundles with independent configurations. Follow these steps to upgrade production exports:

1. Move to the timestamped production export directory for the target profile:

```
Set-Location $env:OEMCP_HOME

# Navigate to production export directory
```

```
# Replace with actual timestamped directory name
Set-Location 'exports\export_<profile>_<timestamp>'
```

**Note:** This command selects the first matching export. If you have multiple exports for the same profile, specify the exact directory explicitly.

2. Stop the running container and back up the export compose file:

```
# Stop and back up
.\run.ps1 stop -Port <port> -Name <container-name>
Copy-Item 'docker-compose.mcp.yml' 'docker-compose.mcp.yml.backup'
```

3. Confirm the export does not include disallowed private credentials:

```
# Verify production secret handling
Test-Path 'sa\private.pem'  # Expected: False
Test-Path 'sa\sa.jwt'       # Expected: False
Test-Path 'sa\public.pem'   # Expected: True
```

4. Load the target images:

```
# Load new images and start
Set-Location $env:OEMCP_HOME
.\load_images.ps1
```

**Note:** This procedure assumes that the OpenEdge MCP Server distribution is available on the production machine.

5. Restart the production export with the upgraded image:

```
# Start the upgraded container
Set-Location 'exports\export_<profile>_<timestamp>'
.\run.ps1 start -Port <port> -Name <container-name>
```

6. Verify the container image and runtime status:

```
# Verify container uses the upgraded image
$ContainerImageId = docker inspect --format '{{.Image}}' <container-name>
$ExpectedImageId = docker image inspect openedge-mcp-server:latest --format '{{.Id}}'

if ($ContainerImageId -eq $ExpectedImageId) {
    Write-Host "✓ Container is using the upgraded image" -ForegroundColor Green
} else {
    Write-Host "✗ WARNING: Container image mismatch" -ForegroundColor Red
    Write-Host "  Container: $ContainerImageId"
    Write-Host "  Expected:  $ExpectedImageId"
}
# Check runtime status
.\run.ps1 status -Port <port> -Name <container-name>
```

Verify that externally managed credentials can list and invoke authorized tools. Production bundles must never receive copied development credentials.

## Step 6: Pin explicit image references (Optional)

Pinning to explicit image versions provides greater control over which version runs in your deployment. Follow these steps to pin an explicit image reference:

1. Set the desired image tag or digest:

```
$ServerImageReference = 'openedge-mcp-server:<version>'
```

2. Apply the image reference to the generated profile compose file:

```
# Update profile compose file
$ComposePath = "generated\$Profile\docker-compose.mcp.yml"
(Get-Content -LiteralPath $ComposePath) `
  -replace '^(\s*)image:.*$', ('${1}image: ' + $ServerImageReference) |
  Set-Content -LiteralPath $ComposePath
```

3. Apply the same image reference to the production export compose file:

```
$ExportComposePath = 'exports\export_<profile>_<timestamp>\docker-compose.mcp.yml'
$UpdatedContent = Get-Content -LiteralPath $ExportComposePath
$UpdatedContent = $UpdatedContent -replace '^(\s*)image:.*$', ('${1}image: ' +
$ServerImageReference)
Set-Content -LiteralPath $ExportComposePath -Value $UpdatedContent
```

4. Recreate the container to apply the image reference change:

   - For standard profiles:

```
Set-Location $env:OEMCP_HOME
.\mcpgen.ps1 stop $Profile
.\mcpgen.ps1 start $Profile

# Verify the container uses the pinned image
docker inspect --format '{{.Config.Image}}' "$Profile"
```

   - For production exports:

```
Set-Location 'exports\export_<profile>_<timestamp>'
.\run.ps1 stop -Port <port> -Name <container-name>
.\run.ps1 start -Port <port> -Name <container-name>

# Verify the container uses the pinned image
docker inspect --format '{{.Config.Image}}' <container-name>
```

**Note:** Editing the compose file does not change already-running containers. The container must be stopped and restarted for the new image reference to take effect.

## Step 7: Rollback procedures

If validation identifies issues with the upgrade, you can restore your previous working deployment using one of these rollback methods.

### Image-only rollback

Follow these steps to restore the previous Docker image while keeping the existing profile configuration unchanged:

1. Stop the running profile:

```
Set-Location $env:OEMCP_HOME
.\mcpgen.ps1 stop $Profile
```

2. Reload the previous image:

```
# Reload previous version image
& 'C:\path\to\previous-distribution\load_images.ps1'
```

3. Restart the profile:

```
.\mcpgen.ps1 start $Profile
```

## Complete profile rollback

Follow these steps to restore the profile directory from backup and return the deployment to its previous working state:

1. Stop the running profile:

```
Set-Location $env:OEMCP_HOME
.\mcpgen.ps1 stop $Profile
```

2. Move the current generated profile aside for analysis:

```
# Preserve upgraded profile
Move-Item "generated\$Profile" "generated\$Profile.post-upgrade"
```

3. Restore and restart the profile:

```
# Restore from backup
Expand-Archive -LiteralPath "..\$Profile-backup-<timestamp>.zip" `
  -DestinationPath . -Force

.\mcpgen.ps1 start $Profile
```

4. Restore the previous mcpgen wheel files:

```
# Restore wheel files from the previous distribution
Copy-Item 'C:\path\to\previous-distribution\wheels\mcpgen-*.whl' `
    "$env:OEMCP_HOME\wheels\" -Force
```

**Note:** If you removed the previous distribution, you can restore the mcpgen tooling by reinstalling from the archived distribution zip file.

### Production bundle rollback

Follow these steps to restore production export artifacts and, when needed, the previous container image:

1. Stop the production container:

```
Set-Location 'exports\export_<profile>_<timestamp>'
.\run.ps1 stop -Port <port> -Name <container-name>
```

2. Restore the compose file:

```
# Restore previous compose file
Copy-Item 'docker-compose.mcp.yml.backup' 'docker-compose.mcp.yml'
```

3. Reload previous image (if using latest):

```
# If using :latest, reload previous image
& 'C:\path\to\previous-distribution\load_images.ps1'
```

4. Restart the production container:

```
.\run.ps1 start -Port <port> -Name <container-name>
```

# Post-upgrade best practices

Following these practices after a successful upgrade helps ensure long-term stability and prepares you for future upgrades.

After completing the upgrade and validation:

1. **Monitor runtime behavior**: Watch for unexpected errors, performance changes, or authorization issues in the first 24-48 hours.
2. **Retain backups**: Keep pre-upgrade backups for at least one release cycle.
3. **Clean up development artifacts**: After confirming stability, remove outdated wheel files and previous distribution archives.
4. **Review logs**: Check container logs for warnings about deprecated settings or upcoming breaking changes.
5. **Test edge cases**: Verify error handling, timeout behavior, and authorization enforcement beyond basic tool invocation.
6. **Update monitoring**: Adjust monitoring thresholds if default values changed.
7. **Document lessons learned**: Record any unexpected issues or additional steps needed for future upgrades.

# Troubleshoot OpenEdge MCP Server upgrade

This section provides solutions to common issues that may occur during or after the upgrade process. Use this table to quickly diagnose common upgrade issues and apply targeted fixes:

| Issue | Symptom | Possible causes | Resolution |
|---|---|---|---|
| Profile fails to start | Profile does not start after upgrade | • Configuration incompatibility with new version<br>• Missing or invalid certificates<br>• Port conflicts with other services<br>• Insufficient Docker resources | 1. Check container logs: `docker logs <container-name>`.<br>2. Verify configuration file syntax.<br>3. Confirm certificates are valid and accessible.<br>4. Check for port conflicts: `docker ps`. |
| Tools not discovered | Tool catalog is empty or missing expected tools after upgrade | • OpenAPI source connection failed<br>• Service-account authorization changed<br>• Middleware configuration error | 1. Verify connectivity to OpenAPI source.<br>2. Check service-account token validity.<br>3. Review middleware configuration.<br>4. Examine server logs for connection errors. |
| Authorization failures | Tool invocations fail with authorization errors | • Service-account key or token mismatch<br>• Changed authorization requirements<br>• Certificate validation failures | 1. Verify service-account public key matches private key.<br>2. Regenerate JWT token if format changed.<br>3. Check certificate validity and trust chain. |
| Performance degradation | Tool invocations are slower than before upgrade | • Changed default timeout values<br>• New middleware overhead<br>• Resource contention | 1. Review timeout configuration changes.<br>2. Monitor container resource usage.<br>3. Check downstream service performance.<br>4. Adjust timeout values if necessary. |

# IV

# Observability, logging, and troubleshooting

For details, see the following topics:

- Observability and logging in OpenEdge MCP Server
- Troubleshooting reference
- Frequently asked questions

# 5

# Observability and logging in OpenEdge MCP Server

Observability is the ability to measure and understand the internal state of a system based on the data it produces. It provides insight into how a system behaves under different conditions. It helps answer questions such as:

- Is the system performing as expected?
- Where are potential bottlenecks or failures occurring?
- How can performance be improved?

Observability in the OpenEdge MCP Server includes three key components:

- **Logging**—Captures system and application messages for troubleshooting and analysis.
- **Audit events**—Records security and compliance-related activities.
- **Timing data**—Tracks performance metrics and execution times for operations.

Here are the primary objectives for implementing observability. These objectives ensure that the data collected is useful, secure, and easy to interpret.

- **Provide actionable, low-noise telemetry**—Collect meaningful data that supports security, performance, and usage analysis without overwhelming users with unnecessary details.
- **Enable flexible log formats**—Allow easy switching between human-readable logs and structured logs that can be parsed by machines for automation and analysis.
- **Protect sensitive information**—Prevent exposure of credentials such as service account tokens and private keys while maintaining sufficient traceability for troubleshooting and audits.

For details, see the following topics:


- Logging architecture overview
- Configure logging and related middleware
- Logging modes and usage guidelines
- Audit events and sensitive data handling
- Configure timing middleware
- Correlation and traceability
- Integration with log aggregators
- Debug for rapid issue diagnosis

# Logging architecture overview

The design and organization of components that capture, store, and manage log data within a system form the logging architecture. A typical logging architecture includes several layers, each with a specific responsibility.

The following table summarizes the layers in the logging architecture and their functions:

| Layer | Responsibility |
|---|---|
| Logging middleware | Captures per-request details such as start and stop times, method name, duration, status, and optional payloads. |
| Audit middleware | Records security and policy-related events, including rate limits and authentication decisions from other modules. |
| Timing middleware | Provides fine-grained latency measurements for operations. This feature is optional and can be enabled as needed. |
| Rate limiting wrapper | Emits audit events indicating whether requests are allowed or denied based on rate-limiting policies. |
| Startup inventory | Generates a list of tools, prompts, and scopes once during system startup. |

# Configure logging and related middleware

Logging and timing features can be configured using `runtime` settings within the `mcp_server_config.json` configuration file. These options control log format, payload inclusion, and method filtering.

### Logging configuration keys

The configuration keys in the `runtime.middleware` section control how logging and timing behave in the OpenEdge MCP Server. These settings allow administrators to enable or disable features, choose log formats, and decide what level of detail is captured during request processing. The following table describes the configuration keys for `runtime.middleware.logging`:

| Key | Type | Default value | Recommended value in production | Description |
|---|---|---|---|---|
| `enabled` | Boolean | `true` | _ | Turns request logging on or off. |
| `structured` | Boolean | `false` | `true` | When set to true, outputs logs as JSON objects for easier integration with log aggregators. |
| `include_payloads` | Boolean | `false` | `false` (enable temporarily for debugging) | Includes truncated request and response payload excerpts. Avoid enabling in production to prevent sensitive data exposure. |
| `methods` | `List[str]` (List of strings) | Unset | _ | Restricts logging to specific HTTP verbs or internal method tags. If unset, all methods are logged. |

The following example shows how to enable structured logging and timing:

```
{
  "runtime": {
    "middleware": {
      "logging": {
        "enabled": true,
        "structured": true,
        "include_payloads": false
      },
      "timing": { "enabled": true },
      "rate_limiting": { "enabled": true, "max_requests_per_second": 5 }
    }
  }
}
```

**Note:** Do not enable `include_payloads` in production because payloads may contain sensitive data.

## Logging modes and usage guidelines

Logging can be configured in plain text or structured JSON format. Each mode has advantages and limitations.

The following table compares plain text and structured JSON logging:

| Mode | Advantage | Limitation | Recommended usage |
|---|---|---|---|
| Plain text | Easy to read locally | Harder to parse centrally | Local development and quick demos |
| Structured JSON | Machine searchable and index-friendly | Slightly larger and less readable raw | Staging and production environments |

The following example shows a structured log entry:

```
{
  "ts": "2025-10-05T12:34:56.789Z",
  "level": "info",
  "component": "request",
  "method": "tool.invoke",
  "tool": "listProducts",
  "duration_ms": 123,
  "status": 200,
  "client_id": "abc123",
  "session_id": "s-9f2",
  "payload_truncated": true
}
```

# Audit events and sensitive data handling

Audit events provide traceability for security and policy decisions. Sensitive data must be handled carefully to prevent exposure.

### Audit event categories

Audit events are system-generated records that capture security and policy-related actions within the OpenEdge MCP Server. They provide traceability for critical decisions such as rate limiting, authentication, and system startup. These events are essential for compliance and troubleshooting. The following table lists audit event categories and their triggers:

| Category | Trigger | Notes |
|---|---|---|
| `rate_limit` | Rate limit audit events record decisions made by the rate-limiting middleware. These events help track whether requests are allowed or denied based on configured policies. | When a request is denied, the event includes fields such as status, key, and reason to indicate why the request was blocked. |
| `auth` (future) | Authentication audit events will capture success and failure outcomes. | Centralization of these events is planned for future releases because today they are scattered across multiple modules. |
| `startup_inventory` | Startup inventory audit events provide a snapshot of system configuration at startup. These events help administrators verify which tools and prompts were loaded during initialization. | These events are logged once at boot unless verbose restart mode is enabled, in which case they may appear more frequently. |

### Sensitive data handling guidelines

The following guidelines describe how sensitive data is managed and what steps administrators should take to avoid accidental exposure:

- Service account JSON web tokens (JWT) are masked upstream and never logged.
- Private keys and certificate PEM bodies are not emitted. Only file paths may appear at debug level.
- Set `include_payloads=false` in production to prevent accidental capture of personally identifiable information (PII) or LLM responses.

## Configure timing middleware

Timing middleware is a lightweight component in the request-processing pipeline that measures how long each request takes to execute. When `runtime.middleware.timing.enabled=true`, the MCP Server activates a lightweight middleware that records precise duration metrics for each request. These metrics help administrators and developers analyze latency and identify performance bottlenecks. Timing middleware introduces minimal overhead and is recommended for production environments.

**Best practice:** Combine timing middleware with structured logging to compute latency percentiles such as P50 and P95 externally using log aggregation tools. Performance percentiles are statistical measures that indicate how a system performs for a given percentage of requests. They help you understand not just the average performance but also the distribution of response times across all requests. Average response time can hide outliers. Percentiles provide a clearer picture of user experience by showing how most requests behave and identifying slow responses.

## Correlation and traceability

Correlation and traceability are important for diagnosing issues in multi-service or distributed environments. They allow administrators to follow the flow of related requests across components and identify where delays or failures occur.

The following table lists the correlation fields available in the MCP Server and describes their purpose:

| Field | Purpose |
|---|---|
| `client_id` | Identifies the calling client when provided by the FastMCP context. |
| `session_id` | Correlates a sequence of related tool invocations within the same session. |
| `correlation_id` | Planned for future implementation to support cross-service tracing and provide global correlation across distributed systems. |

## Integration with log aggregators

When structured logging is enabled, each log entry is emitted as a single-line JSON object. This format ensures compatibility with most log aggregation systems and makes ingestion into tools such as Elasticsearch, Loki, Splunk, or OpenSearch straightforward and reliable.

**Note:** Adopt structured logging early. Retrofitting structured logging after deployment results in the loss of valuable historical comparability.

The following steps summarize the recommended pipeline for log aggregation:

1. **Configure log forwarding**—Log forwarding ensures that logs generated by containers or services are collected and sent to a central location for processing. Use one of the following methods to configure log forwarding:

| Method | Description |
|---|---|
| Docker logging driver | Configure the Docker logging driver with the `json-file` format to capture logs directly from containers. |
| Sidecar forwarder | Deploy a sidecar container that forwards logs to the central log aggregator. |

2. **Create a central index**—A central index organizes log data for efficient querying and visualization. Proper indexing improves performance when analyzing metrics such as latency and request rates. Define an index template that maps numeric fields for optimal performance:

| Field | Description |
|---|---|
| `duration_ms` | Represents request duration in milliseconds for latency analysis. |
| `rps` | Indicates requests per second for throughput monitoring. |

3. **Build dashboards**—Dashboards provide a visual representation of key metrics. They help you monitor system health and identify performance bottlenecks quickly. Create dashboards to track the following metrics:

| Metric | Description |
|---|---|
| Request rate | Displays the number of requests processed per second. |
| Median latency | Shows the median response time for requests. |
| Deny counts | Tracks the number of denied requests. |
| Top tools by volume | Lists tools generating the highest log volume. |

# Debug for rapid issue diagnosis

This topic provides a structured approach for diagnosing issues in runtime environments. It includes configuration toggles, visibility settings, and scenario-based troubleshooting steps. The goal is to enable developers and operators to quickly isolate problems without introducing unnecessary log noise or performance overhead.

## Quick enablement matrix

This matrix lists common debugging goals and the corresponding settings or actions to achieve them. It also includes notes on the impact of each setting:

| Goal | Settings | Notes |
|---|---|---|
| See request and tool call flow | `runtime.middleware.logging.enabled=true` | Provides base visibility |
| Add latency metrics | `runtime.middleware.timing.enabled=true` | Low overhead |
| Machine-parseable root logs | `logging.json=true` | Enables JSON for all loggers |
| Structured request events | `runtime.middleware.logging.structured=true` | Independent of root format |
| Enrichment annotation lines | `openapi_logging.annotations=true` | Adds per-operation summary |
| Verbose enrichment detail | `openapi_logging.verbose=true` | Adds raw or truncated size information |
| Audit tool invocations | `runtime.middleware.audit.enabled=true` | Emits one JSON event per call |
| Outbound HTTP decisions | `logging.loggers.http.outbound=DEBUG` | Shows query normalization and retries |
| TLS or mTLS diagnostics | `logging.loggers.http.tls=DEBUG` | Displays certificate and key load and verification |
| Response guard actions | `logging.loggers.response_guard=INFO` | Logs block, trim, or warn events |
| Downstream client config keys | `logging.loggers.http.config=DEBUG` | Emits config keys and resolved default headers |

## Configure root logging and OpenAPI visibility for debugging

Configure root logging and OpenAPI visibility settings to improve observability during debugging. These configurations provide structured logs, JSON output, and enhanced visibility into API specifications and context annotations. Use these settings when you need deeper insights into request flows and operational context without enabling global `DEBUG`.

Use the following configuration examples to enable root logging and OpenAPI visibility.

### Root logging configuration example

This example shows a minimal root logging configuration that enables JSON output and sets log levels for key components:

```
{
  "logging": {
    "level": "INFO",
    "json": true,
    "loggers": {
      "openapi": "INFO",
      "startup": "INFO",
      "http.outbound": "DEBUG",
      "response_guard": "INFO",
      "tool": "INFO",
      "audit": "INFO"
    }
  }
}
```

**Note:** Structured middleware events `runtime.middleware.logging.structured=true` can coexist with root JSON logging.

### Debug OpenAPI specification visibility

This configuration enables visibility into OpenAPI specifications and context annotations:

```
{
  "openapi_visibility": {
    "add_filter_examples": true,
    "add_param_examples": true,
    "per_tool_context_enabled": true,
    "context_dir": "./context",
    "global_context_char_limit": 4000
  },
  "openapi_logging": { "annotations": true, "verbose": false }
}
```

**Note:** Look for OpenAPI logger lines that confirm global or per-tool context application and tag additions.

## Scenario-based troubleshooting

Use the following guidance that provides targeted troubleshooting steps for common debugging scenarios.

### Per-tool context missing

Perform the following checks to ensure per-tool context is correctly configured. Missing context can prevent proper enrichment and visibility during debugging. For more information about managing per-tool context, see Manage per-tool context on page 223.

1. Confirm that the `context/` directory exists in the container.
2. Verify that `per_tool_context_enabled` is not set to `false`.
3. Ensure files are named using the `<operationId>.md` or `<operationId>.txt` format.
4. Enable `openapi_logging.annotations` and restart the service.

## Large responses trimmed or blocked

Perform the following steps to verify and adjust response guard settings when large responses are trimmed or blocked. These limits help prevent excessive payload sizes but may need adjustment for certain operations.

1. Verify the mode in the logs. The mode can be `block`, `trim`, or `warn`.
2. Inspect the `runtime.http.response_guard` configuration to confirm current limits.
3. Increase `max_items` and `max_bytes` values incrementally to avoid excessive payloads.

## Resolve tool invocation errors

During tool invocation, errors can result from incorrect request handling, parameter validation issues, or downstream header misconfigurations. Perform the following checks to diagnose and resolve tool invocation errors:

1. Turn on structured logging to capture request IDs for correlation.
2. Match the `errors` logger stack trace with the corresponding `audit` event for detailed context.
3. If parameter validation is enabled, review the `tools.validation` logs for errors.
4. Set `logging.loggers.http.config=DEBUG` to confirm that `default_headers` are resolved correctly and that the `Authorization` header is preserved.
5. For repeated `"Invalid params"` errors (JSON-RPC -32602) without enabling global `DEBUG`, set `runtime.middleware.validation_diagnostics.enabled=true` and check `validation.diagnostics` logs for method names and argument key lists.

## Resolve authentication and token exchange issues

Problems related to authentication and token exchange often involve token expiry or parsing errors. The following configuration settings help you enable targeted debugging without exposing sensitive information:

| Setting | Purpose |
|---|---|
| `auth.token_exchange=DEBUG` | Enables detailed logs for token exchange operations |
| `auth.verify=DEBUG` | Provides visibility into token verification steps |

## Diagnose rate limiting denials

Rate limiting denials occur when requests exceed the configured threshold. Audit events provide visibility into these occurrences. Use the following steps to verify rate limiting denials and adjust configuration settings:

- **Check audit events**—Look for entries where `"category":"rate_limit"` appears in the logs.
- **Tune request limits**—Adjust the `max_requests_per_second` setting to align with expected traffic patterns.

## Debug TLS and mTLS issues

Enabling detailed logging for TLS and mTLS operations help diagnose certificate loading and hostname verification problems during secure communication. Enable the `http.tls=DEBUG` setting to display certificate and key loading details and hostname verification decisions.

## Troubleshooting and verification

This topic describes common symptoms observed in log aggregation and provides guidance on what to investigate. Use this information to quickly identify and resolve issues in your logging pipeline.

| Symptom | Investigate |
|---|---|
| Sudden spike in `rate_limit` denies | Misconfigured `max_requests_per_second` or a runaway client loop |
| Missing audit events | Rate limiting disabled or client IDs not set |
| High `duration_ms` | Downstream API slowness, token exchange latency, or TLS handshakes |
| Log volume explosion | Payload logging accidentally enabled with large responses |

### Quick verification commands

You can use the following example commands to confirm the presence of structured logs and to check for rate limit denies. These commands help validate the health of the logging pipeline:

- The following command displays the last 20 log entries:

```
./mcpgen logs <profile_name> --tail=20
```

- The following command counts the number of rate limit denies in the last 500 log entries:

```
./mcpgen logs <profile_name> --tail=500 | grep '"category":"rate_limit"' | grep
'"status":"deny"' | wc -l
```

- The following command scans the last 500 log entries and filters for audit events. Audit logs provide visibility into tool invocations and rate limiting actions.

```
./mcpgen logs <profile_name> --tail=500 | grep '"logger":"audit"'
```

- The following command counts the number of response guard actions in the last 500 log entries. Use it to monitor trimming, blocking, or warning events for large responses.

```
./mcpgen logs <profile_name> --tail=500 | grep '"logger":"response_guard"' | wc -l
```

- The following command checks whether per-tool context annotations have been applied. It helps confirm that OpenAPI visibility settings are working as expected.

```
./mcpgen logs <profile_name> --tail=500 | grep 'Visibility: added per-tool context'
```

## Inspect logs in Kubernetes deployments

You can retrieve and filter logs from Kubernetes deployments for debugging purposes. Use the following commands to inspect logs in Kubernetes environments:

- To view recent logs from the deployment, run:

```
kubectl logs deploy/oemcp-server -n <ns> --tail=100
```

  This command retrieves the last 100 log entries from the `oemcp-server` deployment in the specified namespace. You can use the output to quickly review recent activity and confirm that logs are being generated.

- To stream logs and filter audit events, run:

```
kubectl logs deploy/oemcp-server -n <ns> -f | jq '. | select(.logger=="audit")'
```

  This command streams logs in real time and uses `jq` to filter entries where the logger is set to `"audit"`. Audit logs provide visibility into tool invocations and rate limiting actions.

## Configure a minimal debug profile for temporary use

You can apply a minimal debug profile for short-term troubleshooting. This configuration enables detailed logging and structured middleware events without introducing excessive noise. Use this profile only in non-production environments or for temporary debugging sessions.

Use the following configuration to enable a minimal debug profile:

```
{
  "logging": {
    "level": "DEBUG",
    "json": true,
    "loggers": {
      "http.outbound": "DEBUG",
      "openapi": "DEBUG"
    }
  },
  "runtime": {
    "middleware": {
      "logging": {
        "enabled": true,
        "structured": true,
        "include_payloads": false
      },
      "timing": {
        "enabled": true
      }
    }
  },
  "openapi_logging": {
    "annotations": true,
    "verbose": true
  }
```

Add `"http.config": "DEBUG"` under `logging.loggers` to inspect the downstream HTTP client configuration. This setting displays configuration keys and the final resolved default headers.

## Enable targeted diagnostics for invalid parameters

You can enable targeted instrumentation for HTTP 400 errors caused by invalid parameters. This configuration provides deeper visibility into parameter validation failures without introducing broad `DEBUG` noise.

To enable targeted diagnostics for invalid parameter errors, apply this configuration:

```
{
  "runtime": {
    "middleware": { "validation_diagnostics": { "enabled": true } }
  },
  "logging": {
    "loggers": { "validation.diagnostics": "INFO" }
  }
}
```

The following log line appears:

```
{"level":"info","logger":"validation.diagnostics",
"msg":"invalid_params: method=tool.invoke
message=Invalid params: missing required field 'id' arg_keys=['filter','top']"}
```

Here are the important details about how invalid parameter diagnostics are sanitized to protect sensitive information. Understanding these santization aspects helps you interpret logs correctly and maintain security.

| Aspect | Description |
|---|---|
| Logged content | Only the error message text (truncated to 300 characters) and the list of top-level argument keys |
| Excluded content | No raw argument values are logged |
| Recommendation | Revert the logger level to `INFO` after completing diagnosis |

## Reduce log noise after debugging

Excessive logging can impact performance and make it harder to identify critical issues. Use the following recommendations to replace broad logging settings with targeted alternatives:

| Remove | When | Replace with |
|---|---|---|
| `Global DEBUG` | After fixing issues | Targeted per-logger DEBUG |
| `include_payloads=true` | In production | Short-lived redaction audits |
| `http.outbound DEBUG` | For stable APIs | `INFO` level logging combined with external APM |

## Add custom loggers for extended debugging

You can extend debugging capabilities by creating custom loggers for new modules by:

- Using `logging.getLogger("module.sub")` to create a logger for a new module or subcomponent.
- Adding a configuration key for verbose loops, such as `debug_dump`, to capture detailed execution data when required.

## Triage checklist before filing issues

Capturing essential debugging information before raising an issue ensures faster resolution and reduces back-and-forth during troubleshooting. Confirm the following items are collected and validated before filing an issue:

- A structured log sample window has been captured.
- A matching `audit` event is present in the logs.
- Response guard actions have been confirmed or ruled out.
- Per-logger overrides have been listed.
- Enrichment annotation lines have been captured if the issue is context-related.

# 6

# Troubleshooting reference

This section describes common issues that may occur when using the MCP tools and provides guidance on diagnosing and fixing them. Use this information to troubleshoot errors quickly and maintain smooth operations.

The following table lists symptoms, possible diagnoses, and recommended fixes:

| Symptom | Diagnosis | Fix |
|---|---|---|
| 401 Missing service account | Service account required but header is absent or invalid | Add `X-OEMCP-SERVICEACCOUNT` header with a valid JWT token |
| Empty tool list | Scope filtering or authentication mismatch | Ensure authentication is enabled and token has required scopes. Disable filtering if necessary |
| Large array rejection | Response guard blocked large arrays | Increase `max_array_items` or switch to pagination |
| TLS handshake failure | Incorrect certificate or key path, or client trust store | Recreate certificates using `--tls` option and verify file permissions |
| Certificate hostname mismatch | Client accessing server with a hostname not in certificate | Use `--extra-hostnames` to include all expected domains in the certificate |
| Export bundle missing files | Wrong bundle type or corrupted export | Check `--bundle` type. Use `dev` for full artifacts and `prod` for production bundles |
| Service account token not found | Named token does not exist or incorrect name | Use `./mcpgen sa-tokens <profile>` to list available tokens |
| OpenAPI generation failed | Invalid catalog source or network issue | Verify catalog file or URL accessibility and format |
| Token exchange timeout | Exchange URL unreachable | Check network connectivity and ensure container can resolve the host |
| High latency on first call | JWKS fetch delay | Warm up cache by invoking a simple tool at startup |
| Unclear HTTP errors or authentication issues | Lack of detailed request or response information | Use `./mcpgen client <profile> --debug` to view HTTP details |
| Tool invocation fails with `Internal error`. | The server encountered an internal failure and cannot complete the tool call. | Review the server log files to identify the cause. Run the following command:<br>`docker compose -f docker-compose.mcp.yml logs -f --tail=50` |

| Symptom | Diagnosis | Fix |
|---|---|---|
| Smoke test fails when running `./mcpgen up` or `./mcpgen restart`. For example, `call_tool` fails with error `Internal error` or `ValidationError`. | The automated smoke test is unable to validate tool access during startup or restart. | Do not ignore smoke test failures. Investigate the cause before bypassing the smoke test. After investigating, if you have confirmed a valid reason for the failure, such as a timeout, or are unable to determine the cause, you may proceed with the following workaround:<br>**1.** Rerun `./mcpgen up` or `./mcpgen restart` with the `--no-smoke` option, for example:<br>`./mcpgen up --no-smoke`<br>`./mcpgen restart --no-smoke <container_name>`<br>**2.** Manually call a tool to verify access:<br>`./mcpgen client --tool <tool_name> --json <profile>`<br>If this call fails, such as with an `Internal error`, see the corresponding entry in this table. |
| The smoke test fails and times out when calling the data service during the `./mcpgen restart`, `./mcpgen start`, or `./mcpgen up`. | Smoke test is failing because the request exceeded the middleware request timeout before the backend API returned. | |

| Symptom | Diagnosis | Fix |
|---|---|---|
| | | Proceed with the following workaround:<br>**1.** Increase middleware request timeout in `mcp_server_config.json`:<br>`"runtime": { "middleware": { "request_timeout": { "enabled": true, "timeout_ms": <timeout_value> } } },`<br>**2.** Restart profile container.<br>**3.** If startup must proceed immediately, skip smoke test once:<br>`./mcpgen restart --no-smoke openedge-mcp-server`<br>**4.** Validate connectivity with a direct tool call:<br>`./mcpgen client --tool getCustomer --json openedge-mcp-server`<br>**Note:** The product default request timeout is 30000 ms (30 seconds). Increase only when needed for legitimate slow operations. |

# 7

# Frequently asked questions

This section answers common questions about MCP configuration, security, and operational practices.

**Why are write methods missing by default?**

Write methods are excluded by default for safety. Unless explicitly included, non-GET operations are disabled to prevent unintended data changes.

**Can I hot reload the OpenAPI specification?**

Hot reload is not supported. Restart the server or profile after making changes to the OpenAPI specification. This ensures deterministic tool catalogs.

**How do I add a new helper tool?**

Implement the logic under `src/tools/`. Register the tool in `create_server` following the pattern used for prompt or tag tools. Optionally, add scope requirements for access control.

**Can I disable the response guard?**

Yes, you can disable it by setting `runtime.http.response_guard.enabled=false`. However, this is strongly discouraged in production. Instead, adjust limits to handle large responses safely.

**Is payload logging safe?**

Payload logging can expose sensitive data. Keep `include_payloads=false` outside controlled debugging environments.

**How do I run multiple profiles simultaneously?**

Assign unique host ports using the `--port` option while using up command for profile startup. Each profile will have its own directory and container name.

**Why export instead of copying the generated folder directly?**

Export normalizes content, pins image digests, strips development artifacts, and creates a consistent deployment package suitable for production.

**Does JWKS support algorithm pinning?**

Yes. Use `security.authentication.service_account.algorithms` to restrict accepted signing algorithms.

**How do I test mTLS quickly?**

Generate certificates with `--mtls` and use the bundled client:

```
./mcpgen client <name>
```

For external clients, mount `certs/dev/client.pem` and the key according to the TLS options of your tool.

**How do I avoid certificate hostname warnings?**

Use the `--extra-hostnames` flag when generating certificates to include all domains your clients will use.

Examples:

```
Single domain:
./mcpgen up --quick --tls --extra-hostnames api.example.com

Multiple domains:
./mcpgen up --quick --tls --extra-hostnames
"api.example.com,staging.example.com,myapp.local"

For mTLS:
./mcpgen up --quick --mtls --extra-hostnames "api.example.com,staging.example.com"
```

The generated certificates include both `localhost` and your specified hostnames as Subject Alternative Names (SAN) to prevent SSL warnings when clients connect using different hostnames.

**What is the difference between export bundles, such as development, production, strictly-production?**

The differences are as follows:

| Bundle type | Description |
|---|---|
| Development | Includes all artifacts such as context, prompts, private keys, and tokens for development convenience |
| Production | Production-safe bundle with context and prompts, excludes private keys and tokens |
| Strictly-production | Minimal bundle with no context, prompts, private keys, or tokens. Production cleanup applied |

For most production deployments, use:

```
./mcpgen export <profile_name> --bundle prod
```

**What is the difference between export and export-tar?**

Both `export` and `export-tar` create deployment bundles with the same content options and support the same `--bundle` options for development, production, and strictly-production. Here are some differences:

| Command | Description |
|---|---|
| `export` | Creates a directory structure suitable for Docker Compose deployment |
| `export-tar` | Creates a compressed tarball for easier distribution and archival |

Use `export-tar` when transferring between systems or for automated deployments.

**How do I manage multiple service account tokens?**

Use the `sa-token` and `sa-tokens` commands for token management as follows:

```
Create named token:
./mcpgen sa-token <profile_name> --name admin --scopes api.read api.write

List all tokens:
./mcpgen sa-tokens <profile_name> --json

Set primary token:
./mcpgen sa-token <profile_name> --name admin --set-primary

Remove token:
./mcpgen sa-token <profile_name> --name admin --remove
```

**Can I generate OpenAPI specs from OpenEdge Business Entities?**

Yes. Use the `openapi-gen` command:

```
From local catalog:
./mcpgen openapi-gen <catalog.json> -o <openapi.yml>

From remote endpoint:
./mcpgen openapi-gen http://localhost:8810/web/oemanager/catalog -o <openapi.yml>
```

The `openapi-gen` command generates complete OpenAPI 3.0.3 specifications based on Business Entity definitions.

**What should I do if a smoke test fails when running the `./mcpgen up` or `./mcpgen restart` commands?**

Do not ignore smoke test failures. Investigate the cause before bypassing the smoke test. After investigating, if you have confirmed a valid reason for the failure, such as a timeout, or are unable to determine the cause, you may proceed with the following workaround:

1. Rerun the command with the `--no-smoke` option, for example:

```
./mcpgen up --no-smoke
```

```
./mcpgen restart --no-smoke <container_name>
```

2. Manually call a tool to verify access:

```
./mcpgen client --tool <tool_name> --json <profile>
```

   If this call fails, see the corresponding entry in the Troubleshooting reference on page 105 table.

**How do I prepare a profile for production deployment?**

Use the `harden` command to remove development artifacts:

```
Remove client tools:
./mcpgen harden <profile_name> --remove-client

Remove private keys:
./mcpgen harden <profile_name> --drop-private

Remove tokens:
./mcpgen harden <profile_name> --drop-token

Full hardening:
./mcpgen harden <profile_name> --remove-client --drop-private --drop-token
```

**How do I debug HTTP communication issues?**

Use the `--debug` flag with the client command:

```
./mcpgen client <profile_name> --debug
./mcpgen client <profile_name> --tool myTool --args '{"id":123}' --debug
```

Debug output includes:

- Full Docker command executed
- HTTP request details (method, URL, headers with sensitive values masked)
- HTTP response details (status code, headers, body preview)
- MCP protocol message flow
- Timing information

**Note:** Always redact sensitive information before sharing logs.

**How do I use different service account tokens for testing?**

The `client` command supports named token selection:

```
./mcpgen client <profile_name> --sa-token admin --tool sensitiveOperation
./mcpgen client <profile_name> --sa-token readonly --tool listData
```

The `client` command allows testing with different permission levels without recreating profiles.

**How do I rotate service account keys completely?**

Use `remint-all` to generate a new key pair and re-mint all tokens:

```
./mcpgen remint-all <profile> --days 90 --scopes mcp_access api.read
```

This command creates fresh keys and regenerates all existing tokens with the new keypair.

**Are SSE and streamable HTTP interchangeable?**

Yes. Choose based on client capability. Streamable HTTP is the default for simplicity. Enable SSE through configuration or the `MCP_USE_SSE` environment variable.

**Does rate limiting apply per tool?**

Rate limiting is currently applied per inbound request, which means each tool invocation is treated as an individual request. If you need to distinguish between tools, you can review the server logs for detailed information.

**Are denies cached?**

Denials are not cached. Each incoming request consumes or checks the bucket state in real time. This ensures accurate enforcement of rate limits without relying on cached decisions.

**Can I exempt a service account?**

Service account exemption is *not* supported yet. The roadmap includes an optional scope-based exemption field that will allow administrators to configure exemptions in future releases.

**Do I need both `summary` and `description`? in openAPI specification?**

Yes. The `summary` provides a concise label for the operation, while the `description` supplies actionable context, assumptions, and edge cases.

**Can I omit the `title` in schemas?**

Avoid omitting the `title`. It improve readability in helper tools and catalog listings. Including a title makes schemas easier to identify and reuse.

**Should I inline trivial response objects in API documentation?**

Prefer components even for small shapes if reused more than once. Using components deduplicates definitions and clarifies intent. Inline objects should only be used when they are unique and not repeated.

**How do I distinguish partial updates?**

Use the `PATCH` method with a separate schema or `application/merge-patch+json`. Document the semantics clearly so consumers understand how partial updates differ from full replacements.

**Must every schema property declare a `type`?**

Yes. Omitting the `type` prevents reliable parsing and weakens validation. Always specify at least the type and include the format when meaningful. This practice strengthens validation and improves prompt shaping.

**Can I redact specific fields selectively in logs?**

Not yet supported. The recommended workaround is to disable `include_payloads` and rely on upstream sanitization to remove sensitive data before logs are ingested.

**Does structured logging mode guarantee ordering?**

Ordering is preserved per container `stdout` only. There is no global ordering across replicas.

**Are multi-line logs emitted?**

No. Each event is emitted as a single-line JSON record. This design simplifies ingestion and ensures compatibility with log aggregation tools.

## Escalation checklist before filing an issue

Before raising a support ticket, complete these steps:

- Run `./mcpgen doctor` to verify system dependencies and configuration.
- Capture the failing tool invocation including inputs, truncated output, and correlation ID.
- Confirm configuration matches documented properties and does not contain typos.
- For authentication issues, specify whether a service account is required, include JWKS settings, and provide a sample (with sensitive data redacted) JWT header.
- Use the `--debug` flag for detailed HTTP communication logs when applicable.

## How to collect diagnostics

Follow these steps:

1. Run the following commands to gather diagnostic information:

```
./mcpgen check <profile_name>
./mcpgen logs <profile_name> --tail=200 > issue.log
```

2. Redact secrets such as tokens and private keys.
3. Attach a sanitized `mcp_server_config.json` file to the support ticket.

## Use the --debug flag for HTTP troubleshooting

The `mcpgen client` command supports a `--debug` flag that shows detailed HTTP request and response information:

```
./mcpgen client <profile_name> --debug
./mcpgen client <profile_name> --tool listProducts --args '{"limit":3}' --debug
```

Use `--debug` when diagnosing:

- Authentication or authorization failures
- Unexpected API responses
- Header propagation issues
- Downstream HTTP communication problems
- Token exchange issues
- SSL or TLS handshake problems

**Note:** Always redact sensitive information, including tokens, keys, PII, and so on, before sharing logs.

## Targeted 400 invalid params instrumentation

If you encounter repeated `Invalid params` errors without a clear cause, follow these steps:

1. Enable the diagnostics middleware:

```
{
  "runtime": { "middleware": { "validation_diagnostics": { "enabled": true } } },
  "logging": { "loggers": { "validation.diagnostics": "INFO" } }
}
```

2. Reproduce the failing call and inspect logs for lines like:

```
invalid_params: method=tool.invoke message=Invalid params: missing field 'id'
arg_keys=['filter','top']
```

3. Disable diagnostics afterward by setting `enabled=false` to reduce noise.

## CLI exit codes quick reference

The following table lists the exit codes and their meanings:

| Code | Meaning |
|---|---|
| 0 | Success |
| 1 | Generic error or configuration validation failure |
| 2 | Missing profile or resource |

Non-zero codes indicate failure. Scripts should treat any non-zero exit code as an error.

# V

# Operate, secure, and manage the OpenEdge MCP Server

For details, see the following topics:

- Use the mcpgen CLI for profile generation and management
- Operational guidelines and performance best practices for MCP servers
- Configuration reference
- Security and authentication architecture for MCP server
- Manage prompts and tooling
- OpenAPI visibility and context management
- Export and harden production deployment
- Rate limiting in OpenEdge MCP Server
- Upgrade process and key or certificate rotation

# 8

# Use the mcpgen CLI for profile generation and management

The `mcpgen` Command-Line Interface (CLI) is a comprehensive tool designed to simplify profile generation, configuration management, and lifecycle operations for OpenEdge MCP environments. It offers a rich set of commands and workflows that support development, testing, and production deployments. With `mcpgen`, you can create and manage profiles that encapsulate critical artifacts such as configuration files, certificates, tokens, and helper scripts, ensuring secure and consistent server instances across environments.

For details, see the following topics:


- Understand profiles and artifacts
- Manage core commands in mcpgen
- Profile management and advanced operations
- Advanced commands
- Quick start commands to create development profiles
- Typical workflow and API discovery
- Error patterns and remedies
- Clean removal of profile
- Debug and diagnostics

# Understand profiles and artifacts

Profiles are essential for managing server instances. Each profile contains all artifacts required to run a server, such as configuration files, certificates, tokens, and helper scripts. Profiles are stored under `generated` folder and tracked in `generated/servers.registry.tsv`.

---

**Note:** Use `mcpgen up` only for development and iterative testing. For any production or pre-production environment, create a hardened export using:

```
./mcpgen export <profile_name> --bundle prod
```

Deploy the exported bundle, not the mutable `generated/<name>` development profile directory. For more information, see Export profiles for production on page 127.

---

# Manage core commands in mcpgen

This section describes the core commands available in the `mcpgen` CLI for creating, managing, and inspecting profiles.

## Create and start a profile

Use the `up` command to generate and optionally start a profile container along with its supporting artifacts.

### Syntax

```
./mcpgen up [options]
```

## Options for the up Command

The following table lists the available options for the up command, describing their purpose and any important usage notes:

| Option | Purpose | Notes |
|---|---|---|
| `--quick` | Non-interactive defaults | This flag skips all prompts and automatically selects a derived profile name if one is not provided. |
| `--interactive` | Guided parameter entry | When this flag is used, the CLI launches an interactive wizard for configuration. It cannot be combined with `--quick`. |
| `--instance NAME` | Specifies an explicit profile or container name. | The name must be unique because it is used as both the directory name and the container identifier. |
| `--port PORT` | Sets the host port mapping for the container. | For example, you can specify `8500`. If omitted, the port is auto-assigned by the system. |
| `--spec PATH` | Provides the path to the OpenAPI specification file. | If this flag is not specified, the CLI defaults to OpenAPI specification file if that file is present in the working directory. |
| `--prompts DIR` | Specifies the path to the prompts directory containing Markdown files. | If this is omitted, the server uses the default prompts directory, which is the `./prompts` directory in the current working directory.<br>Use `--no-prompt` to disable prompt loading entirely. |
| `--tls` | Enables server-side TLS for secure communication. | When this flag is used, development certificates are generated automatically if they do not already exist. |
| `--mtls` | Enables mutual TLS (mTLS) for enhanced security. | This flag implies `--tls` and additionally generates a client CA and client certificate for mutual authentication. |
| `--extra-hostnames HOST1,HOST2,...` | Adds additional hostnames to TLS certificates. | These hostnames are included as Subject Alternative Names (SAN) in the generated certificates. This works with both `--tls` and `--mtls`. |
| `--no-sa` | Skips the creation of service account keys and tokens. | When this flag is used, tool scopes are limited to user authentication only. |
| `--no-prompt` | Prevents loading of prompt markdown files. | Prompt tools remain registered but return empty selections when invoked. |
| `--no-smoke` | Skips the post-start smoke test. | This is useful when troubleshooting startup issues or iterating on failing configurations. |

| Option | Purpose | Notes |
|---|---|---|
| `--images IMAGE[,UTILS]` | Overrides the default server image and optionally the utilities image. | This feature is experimental and may be replaced by environment variables in future versions. |

## Behavior summary

The up command performs the following internal operations when creating a new profile:

- Creates a directory under `generated/<profile_name>/` with the server, client, certificates, and service account subfolders.
- Writes the `mcp_server_config.json` configuration file with all specified settings.
- Optionally performs a smoke test (tool listing) unless the `--no-smoke` flag is used.
- Automatically determines authentication header precedence in the following order: service account prefix → user token prefix → Bearer.

## Interactive mode features

When you use the `--interactive` flag, the CLI launches a guided wizard to help configure authentication and other settings.

The wizard prompts:

```
Enable user_auth pass-through? (y/N)
```

Where,

- **y** (yes)
  - Configures token-based authentication (Bearer, OAuth, JWT, etc.)
  - Sets header name (default: `Authorization`)
  - Sets token prefix (default: `Bearer`)
  - Optionally provides a test token
  - Configures token exchange if needed (URL, body field, response path)
- **N (default)** (No)
  - Does not configure user authentication
  - Useful for public APIs or when authentication is external

## Manual authentication setup

Manual authentication setup allows you to define custom headers or credentials for downstream API calls. This is typically used for:

- Basic authentication
- API keys
- Custom header-based authentication

Use these steps to configure authentication manually after profile generation when the interactive wizard is not sufficient or when custom methods are required.

1. Generate Base64-encoded credentials for basic authentication:

```
# For credentials restuser/password
echo -n "restuser:password" | base64
```

2. You can configure authentication in one of two ways:

   a. **Option A**: Edit the file `generated/<profile_name>/server/mcp_server_config.json` and add the header directly:

```
{
  "runtime": {
    "http": {
      "default_headers": {
        "Authorization": "Basic <Base64-encoded credentials>"
      }
    }
  }
}
```

   **Note:** Hardcoding credentials is not recommended for production environments.

   b. **Option B**: Use environment variables to store sensitive information securely by following these steps:

      a. Edit `generated/<profile_name>/server/mcp_server_config.json` to reference the variable:

```
{
  "runtime": {
    "http": {
      "default_headers": {
        "Authorization": "${DOWNSTREAM_AUTH_HEADER}"
      }
    }
  }
}
```

      b. Add the environment variable to `generated/<profile_name>/server/server.env`:

```
# Add to server/server.env
DOWNSTREAM_AUTH_HEADER=Basic <Base64-encoded credentials>
```

      c. Ensure the environment file is loaded automatically by Docker when starting the container:

- `./mcpgen start <container_name>`
- `generated/<profile_name>/start_server.sh`
- `docker-compose.mcp.yml`

**Note:** Both $VAR and ${VAR} syntax are supported for environment variable expansion.

For more information about the additional configurations options and the generated profile structure, see Step 5: Generate and start a profile on page 49.

## List a profile

Use the `list` command to display all known profiles from the registry and verify their on-disk status. This topic explains the command syntax, available flags, and sample output.

### Syntax

```
./mcpgen list [options]
```

This command retrieves profile information from the registry and optionally performs live inspection of container status.

### Flags for the `list` Command

The following table lists the flags supported by the `list` command, along with their purpose and detailed usage notes.

| Flag | Purpose | Notes |
|---|---|---|
| `--json` | Outputs the profile list in machine-readable JSON format. | Use this flag when integrating with automation tools or scripts that require structured data. |
| `--wide` | Displays extended profile details. | Adds columns for port, TLS mode, mTLS status, OpenAPI spec path, service account status, and prompt configuration. |
| `--profiles` | Prints only profile names, one per line. | Useful for scripting scenarios where you need a simple list of profile names for batch operations. |
| `--status` | Forces a refresh of container status. | Performs a live inspection instead of relying on cached registry data. Recommended when container state may have changed. |

### Sample output

The following example shows the output of the `list` command with the `--wide` flag:

```
NAME      PORT TLS  MTLS  SPEC            SA  PROMPTS  CREATED
myprof    8500 yes  no    openapi.yml     yes yes      2025-10-11T07:22:14Z
```

## Start, stop, and restart profile containers

Use these commands to manage the lifecycle of an existing profile container.

### Syntax

```
./mcpgen start [--no-smoke] <container_name>
./mcpgen stop <container_name>
./mcpgen restart [--no-smoke] <container_name>
```

Where,

- The `start` command launches the container associated with the specified profile. Use `--no-smoke` to skip the automatic smoke test after start.
- The `stop` command halts the running container for the profile.
- The `restart` command stops and then starts the container. Use `--no-smoke` to skip the automatic smoke test after restart.

**Note:** If only one container exists, the name is auto-detected and can be omitted.

## View container logs

Use the `logs` command to inspect container logs for troubleshooting and monitoring.

### Syntax

```
./mcpgen logs <profile_name>
```

Because the log output respects container log retention policies, use this command to investigate errors or unexpected behavior in the container. It accepts profile name as a required argument. You can also include the `-f` parameter to follow logs in real time for continuous monitoring.

### Example

```
./mcpgen logs myprof -f
```

This command invokes the `logs` command in the `mcpgen` CLI to display logs from a profile container. In this example, `myprof` is the name of the profile whose container logs you want to view. The `-f` parameter streams the logs in real time. This is useful for monitoring ongoing activity or troubleshooting issues as they occur.

## Check profile status

The `check` command provides a diagnostic summary of the profile container, including its state, recent log snippet, and environment variables.

### Syntax

```
./mcpgen check <profile_name>
```

This command:

- Confirms whether the container is running.
- Displays a short log excerpt for quick troubleshooting.
- Lists environment variables currently applied to the container.

# Profile management and advanced operations

Profile management and advanced operations in `mcpgen` enable you to transition from development profiles to production-ready deployments and maintain secure, efficient environments. This section explains how to export profiles, harden configurations, generate OpenAPI specifications, and manage service account tokens.

## Export profiles for production

Use the `export` command to bundle a profile for relocation or production deployment. This creates an immutable snapshot required for promotion outside development environments.

### Syntax

```
./mcpgen export <profile_name> [options]
```

The exported runtime profile directory under `exports/<profile_name>/` contains only the essential files required for running the MCP server in production. Development-specific files, such as temporary assets or client convenience scripts are excluded to ensure a clean, secure, and minimal deployment package. After the export, you have:

- A directory named `exports/<profile_name>/` containing the production-ready profile.
- Optionally, a compressed tarball (`profile-<profile_name>.tar.gz`) if you use the `--tar` flag or the `export-tar` command. This tarball is useful for distribution or deployment across environments.

### Flags for the `export` Command

The following table lists the available flags for the `export` command, describing their purpose and any important usage notes:

| Flag | Purpose | Production guidance |
|---|---|---|
| `--output DIR` | Specifies the destination root directory. The default root directory is `./exports`. | Ensure this directory is excluded from version control. |
| `--bundle PRESET` | Defines the content preset: `dev`, `prod`, or `strict-prod`. | Use `prod` or `strict-prod` for production deployments. |
| `--include-private` | Retains private keys in the export. | Avoid unless performing secure backup or migration. |
| `--include-token` | Includes the generated service account JWT. | Rotate the token soon after import if used. |
| `--include-context` | Copies referenced `context/` files. | Ensures enrichment parity across environments. |
| `--include-prompts` | Copies prompt markdown files. | Required if prompts influence workflows. |
| `--prod` | Strips development artifacts and tightens permissions. | Mandatory for any non-development export. |
| `--tar` | Creates a compressed tarball of the export. | Useful for distribution. |

### Export as tarball

Use the `export-tar` command to create a compressed archive of the exported profile.

```
./mcpgen export-tar <profile_name>
```

This command performs an export and packages the profile as `profile-<profile_name>.tar.gz`.

## Harden development profiles

Use the `harden` command to tighten an existing development profile before exporting it for production.

### Syntax

```
./mcpgen harden [options]
```

### Flags for the `harden` Command

The following table lists the available flags for the `harden` command, describing their effect:

| Flag | Effect |
|---|---|
| `--remove-client` | Deletes generated client convenience assets. |
| `--drop-private` | Removes private key material and retains public parts. |
| `--drop-token` | Removes stored service account token. |

### Recommended sequence

To ensure a secure and compliant production deployment, always harden the development profile first and then export the sanitized bundle for production release:

```
./mcpgen harden --drop-private --drop-token
./mcpgen export --bundle prod
```

## Generate OpenAPI specification

Use the `openapi-gen` command to generate an OpenAPI 3.0.3 YAML specification from a Progress Data Object Service catalog, which is in JSON format.

### Syntax

```
./mcpgen openapi-gen <source> [-o <output.yml>]
```

*source*

Progress Data Object Service catalog source, which is either local file path or HTTP(S) URL.

`-o` *output_file*

Output YAML file path. Defaults to an auto-generated name based on source.

### Examples

The following example generates a specification file using a local Progress Data Object Service catalog generated by Progress Developer Studio for OpenEdge for an ABL service:

```
./mcpgen openapi-gen <catalog.json>
```

You can use Progress Developer Studio for OpenEdge to build an ABL Web App with a Data Object Service and generate a catalog. Then provide the URL where that catalog is accessible. The URL to access the catalog should look like this:

```
http://host:port/<web-app>/web/pdo/<ServiceName>
```

For example, if the `Sports2020` web app with the `Sports2020Service` Data Object Service is deployed and available, you can generate the specification file using:

```
./mcpgen openapi-gen http://localhost:8810/Sports2020/web/pdo/Sports2020Service
```

The following example specifies the output YAML file:

```
./mcpgen openapi-gen catalog.json -o sports-api.yml
```

The following example generates the specification file from a remote Progress Data Object Service catalog with a custom output file:

```
./mcpgen openapi-gen https://server.com/web/catalog -o openapi.yml
```

### What `openapi-gen` command does

The `openapi-gen` command performs the following functions when you run it:

- Loads Progress Data Object Service catalog from file or URL. It supports gzip-compressed responses.
- Extracts service metadata, resources, and schema definitions.
- Converts ABL types, such as `INTEGER`, `DECIMAL`, `LOGICAL`, `DATE`, and so on to OpenAPI types.
- Generates CRUD operation paths including, `GET`, `POST`, `PUT`, `DELETE`. For OpenEdge Data Services, the request body schemas for `POST` and `PUT` operations include the expected nested hierarchy, such as dataset and temp-table levels required by the OpenEdge Data Object Service.
- Creates component schemas for request or response structures.
- Outputs clean OpenAPI 3.0.3 YAML with proper formatting.

### Contents of the output specification file

The `openapi-gen` command produces an OpenAPI 3.0.3 YAML file specification file, which contains:

- Service information and server endpoints.
- Component schemas for all data tables.
- CRUD paths for operations including, `GET`, `POST`, `PUT`, `DELETE`. Note that POST, PUT, and DELETE operations are excluded by default when the MCP server loads the specification. To make them available as MCP tools, see Enable write operations (POST, PUT, DELETE) on page 130.
- Query parameters for filtering and pagination.
- Request and response bodies with proper references.

For information about where to place the generated OpenAPI specification in the MCP setup workflow, see "Place the specification" in Step 4: Provide OpenAPI specification on page 48.

## Enable write operations (POST, PUT, DELETE)

When you generate an OpenAPI specification using `openapi-gen`, the output includes CRUD operation paths for GET, POST, PUT, and DELETE. However, the MCP server excludes POST, PUT, and DELETE operations by default when it loads the specification. This means those operations do not appear as MCP tools, even though they are present in the specification file.

This default behavior is intentional. It prevents write operations from being exposed accidentally and keeps the server in a safe, read-only state until you explicitly opt in.

To make write operations available as MCP tools, you must add route configuration to `mcp_server_config.json` and restart the server.

## How write operations are controlled

The MCP server uses routes.maps in mcp_server_config.json to control which operations are exposed as tools. Rules in the maps array are evaluated in order and the first matching rule wins. For more information, see Configure routes on page 161.

The default behavior is equivalent to a global exclusion rule for all mutating methods:

```
{
  "comment": "Global exclusion of mutating methods (default)",
  "methods": ["POST", "PUT", "DELETE", "PATCH"],
  "mcp_type": "EXCLUDE"
}
```

To enable write operations for a specific path, add an opt-in rule with `"mcp_type": "TOOL"` **before** the global exclusion rule.

## Enable write operations for a specific path

The following example enables POST, PUT, and DELETE on the `/Orders` path while keeping all other write operations excluded:

```
{
  "api": {
    "openapi": {
      "spec_file": "openapi.yml"
    },
    "routes": {
      "maps": [
        {
          "comment": "Global exclusion of mutating methods (default)",
          "methods": ["POST", "PUT", "DELETE", "PATCH"],
          "mcp_type": "EXCLUDE"
        },
        {
          "comment": "Opt-in allow POST on /Orders for create operation",
          "pattern": "/Orders",
          "methods": ["POST"],
          "mcp_type": "TOOL"
        }
      ]
    }
  }
}
```

**Note:** Rule order matters. Place specific opt-in rules before the global exclusion rule, otherwise the exclusion rule matches first, and the opt-in rule is never reached.

## Restart the server after changes

After editing `mcp_server_config.json`, restart the server for the changes to take effect:

```
mcpgen restart
```

## Rotate service account keys and tokens

Use the `remint-all` command to generate a new service account keypair and remint all stored service account tokens for the profile in a single atomic operation.

The `remint-all` command is essential for maintaining security by:

- Ensuring every consumer token is re-signed after key compromise or routine rotation.
- Preserving token attributes, such as `name`, `client_id`, `scopes`, `days`, and `issuer` unless manually edited.

### Behavior summary

When executed, the `remint-all` command performs the following actions:

- Creates a fresh private and public key pair under the `sa` directory. The previous keys are overwritten after a successful remint.
- Iterates through the `profile.json.sa_tokens[]` array and re-mints each token using the preserved metadata.
- Updates the timestamp if it is present in the profile.
- Maintains the primary token in the `client/mcp.json` file.

### Exit status condition

The command returns the following exit status conditions:

- Returns a non-zero exit status if any token fails to remint.
- If there are partial failures, the old key pair and tokens remain intact because the operation is designed to be all or nothing.

### Use cases

Use the following commands based on the specific scenario:

| Scenario | Command |
|---|---|
| Adjust scopes on primary token only | `sa-token --set-primary` |
| Key compromise suspected | `remint-all` |
| Quarterly organization-wide key rotation | `remint-all` |
| Need to extend primary token lifetime | `sa-token --set-primary` |
| Add a brand-new token for a new client | `sa-token` |

## Service account multi-token management

Profiles support multiple named service account tokens tracked in `profile.json.sa_tokens`. This enables distinct `client_id` and scope sets, for example CI automation versus read-only dashboards while sharing a single signing keypair.

## Command reference

Use the following commands to manage service account tokens:

| Command | Purpose |
|---|---|
| `sa-token <profile_name>` | Creates or replaces a named token entry. |
| `sa-tokens <profile_name>` | Lists all tokens including names, client IDs, scopes, and primary marker. Add `--json` for machine output. |
| `remint-all <profile_name>` | Rotates the key pair and remint every token. |

## Flags for the `sa-token` command

The following table lists the most commonly used flags, along with their purpose and detailed usage notes. For the full list of flags, run this help command:

```
./mcpgen sa-token -h
```

| Flag | Purpose | Notes |
|---|---|---|
| `--name NAME` | Unique logical token name | The `default` string is reserved for primary generated at profile creation. |
| `--scopes scope1 scope2 ...` | Space-separated scopes | Replaces existing scopes for that token |
| `--client-id ID` | Custom client identifier | Defaults to derived profile-based ID if omitted |
| `--days N` | Validity period in days | Default is 365 |
| `--issuer ISS` | Override issuer | Defaults to profile issuer if set |
| `--primary` | Mark token as primary | Only one primary allowed; updates client header |

## List tokens

The following example command lists all tokens:

```
./mcpgen sa-tokens myprof
```

The command produces the following output:

```
NAME       CLIENT_ID      SCOPES                 PRIMARY  EXPIRES
default    svc-myprof     schema.read tags.read  yes      2026-10-11
ci-bot     ci-bot         orders.read            no       2025-11-10
```

The following example outputs the JSON representation of the `sa_tokens` array from the profile. The `jq` filter then extracts the first token object.

```
./mcpgen sa-tokens myprof --json | jq '.sa_tokens[0]'
```

Here is an example of the `sa_tokens` section inside the `profile.json` file:

```
{
  "sa_tokens": [
    {
      "name": "default",
      "filename": "sa/token.jwt",
      "client_id": "svc-myprof",
      "issuer": "oemcp-local",
      "scopes": ["schema.read", "tags.read"],
      "days": 365,
      "primary": true
    },
    {
      "name": "ci-bot",
      "filename": "sa/ci-bot.jwt",
      "client_id": "ci-bot",
      "issuer": "oemcp-local",
      "scopes": ["orders.read"],
      "days": 30,
      "primary      "primary": false
    }
  ]
```

## Primary token semantics

Only the primary token is embedded into `client/mcp.json` as the service account header. The `--sa-token` <name> flag lets you override the default primary token and select any named token configured in `profile.json.sa_tokens`. This behavior supports least privilege access and role-based workflows.

The following examples show how to override the default token and specify a named token:

- This example runs the MCP client for the profile named `myprof` and uses the service account token named `ci-bot` for authentication. This approach is useful when you want to switch from the default token to a token with different scopes or permissions.

```
./mcpgen client myprof --sa-token ci-bot
```

- This example runs the MCP client for the profile named `myprof`, uses the service account token named `ci-bot` for authentication, and also invokes a specific tool called `getOrders` with empty arguments (`{}`). This approach allows you to run a targeted API operation under the authorization of the `ci-bot` token, ensuring the correct access level for that tool.

```
./mcpgen client myprof --sa-token ci-bot --tool getOrders --args '{}'
```

**Note:** To mark a different token as primary, re-issue it with `--primary`. The previous primary flag is cleared automatically.

## Example procedure to rotate service account tokens

Follow these steps to perform token rotation:

1. To add a new limited-scope token, run the following command to create a token named `dashboard` with the `products.read` scope, a custom client ID, and a validity of 90 days:

```
./mcpgen sa-token myprof --name dashboard
--scopes products.read --client-id dashboard --days 90
```

2. To promote the new token to primary, make the `dashboard` token the primary token for the profile:

```
./mcpgen sa-token myprof --name dashboard --primary
```

3. To rotate all keys and tokens after quarterly review, use the `remint-all` command to generate a new key pair and re-mint all tokens:

```
./mcpgen remint-all myprof
```

4. To verify the updated tokens, list all tokens in JSON format to confirm the changes:

```
./mcpgen sa-tokens myprof --json
```

### Revoke or delete a token

Currently, there is no direct command for token removal. To remove a token:

1. Delete its JWT file.
2. From the `generated/<profile_name>/profile.json` file, remove the `sa_tokens` object.
3. Optionally, run `remint-all` if you also want a key rotation.

### Security considerations

Ensure these recommendations to maintain token integrity and prevent compromise:

- All tokens share the same signing key. A compromise of the private key invalidates security for every token. Use `remint-all` promptly.
- Distinct scopes per token support least privilege and simplified audit trails in external systems.

## Cleanup and lifecycle management commands

The following commands help you manage container lifecycle and clean up resources:

| Command | Description |
|---|---|
| `prune` | Removes stopped containers and orphaned profile directories.<br>**Note:** Add `--dry-run` to preview the changes before applying them. |
| `terminate` | Stops *all* containers and removes profile directories.<br>**Caution:** This is a destructive operation intended for environment reset. |
| `repair-registry` | Rebuilds the registry file from directories. This command is useful if manual deletions occur.<br>**Note:** Add `--force` to override an existing registry. |
| `stop-all` | Stops running containers while preserving directories. |

## Diagnostics and inspection commands

The `mcpgen` CLI provides a set of diagnostic and inspection commands designed to help administrators and developers validate the environment, inspect profile metadata, interact with tools exposed by a profile, and more. Together, these commands provide a structured approach to diagnosing issues, verifying readiness, and ensuring smooth integration with OpenEdge MCP workflows.

### Doctor

Use the `doctor` command to run environment and dependency diagnostics. It validates the setup and checks for required components. For Compose, it first looks for plugin v2 of `docker compose` and falls back to legacy `docker-compose` if present.

Here is a sample abbreviated output:

```
[doctor] docker: OK
[doctor] docker server version: 26.1.1
[doctor] docker compose (v2): OK
[doctor] python   : 3.12.3 (supported)
[doctor] openedge-mcp-server: present
[doctor] openedge-mcp-utils  : present
[doctor] network (mcpnet): exists
[doctor] done
```

Where,

- `(supported)`—Specifies that version is within 3.11–3.13, which is the validated range.
- `(unsupported — need 3.11–3.13)`—Specifies that version is outside validated range. Upgrade or downgrade is recommended.

Running `doctor` early shifts environment errors left, enabling faster time to first successful profile.

## Inspect

Use the `inspect` command to print the full `profile.json` metadata. The metadata includes specification path, container name, TLS mode, creation timestamp, and more.

## Client

The `client` command provides a containerized client convenience wrapper for tool discovery, invocation, and debugging. Use the following patterns to list tools, view details, and invoke operations using the `mcpgen client` command.

| Action | Command |
|---|---|
| List tools with descriptions | `./mcpgen client <profile_name>` |
| Show full tool details | `./mcpgen client <profile_name> --list-details` |
| Show details for a specific tool | `./mcpgen client <profile_name> --list-details --tool getInventoryRecords` |
| Invoke a tool | `./mcpgen client <profile_name> --tool listProducts --args '{"limit":3}'` |
| Invoke a tool with pretty JSON args through a file | `./mcpgen client <profile_name> --tool getOrders --args @args.json` |
| Invoke a tool with args from file (explicit flag) | `./mcpgen client <profile_name> --tool getOrders --args-file args.json` |
| List tools in JSON format | `./mcpgen client <profile_name> --list-json` |
| Raw tool output (no automatic JSON parse) | Append `--raw` (if implemented) |
| Debug HTTP requests/responses | Append `--debug` to see detailed HTTP communication |

## Options for the `client` Command

Use the following table to understand the purpose of each option and see examples of how to use them with the `client` command:

| Option | Purpose | Example |
|---|---|---|
| `--tool NAME` | Invoke a specific tool by name, or filter tools | `--tool listProducts` |
| `--args JSON` | Provide tool arguments as a JSON string or file | `--args '{"limit":3}'` or `--args @args.json` |
| `--args-file PATH` | Read tool arguments from a JSON file | `--args-file args.json` |
| `--sa-token` | Use a specific service account token by name | `--sa-token admin` |
| `--raw` | Output raw tool response without JSON formatting | For parsing in scripts |
| `--list-json` | Output tools with names and descriptions in JSON | For programmatic use |
| `--json` | Suppress all non-JSON informational output; stdout contains only valid JSON | For use with automation and CI pipelines; safe to pipe the output to JSON parser such as `jq` |
| `--list-details` | Show full tool details including schema and examples | For API exploration; combine with `--tool` |
| `--debug` | Display detailed HTTP request and response info | For troubleshooting HTTP issues |

## Examples

- When you run `./mcpgen client <profile>` without specifying a tool, the following output is displayed:

```
     Tools: 12 |    Prompts: 2
ToolNames:
- getInventoryRecords: Retrieve standard inventory stock levels and availability
- getShippingRecords: Retrieve order fulfillment and delivery tracking records
- getSupportTickets: Retrieve customer service tickets and issue tracking records
...
```

  Each tool is shown with its name and description to help you understand what it does.

- When you run `./mcpgen client <profile> --list-details`, the output displays beautified metadata for each tool:

  - To show all tools:

```
./mcpgen client <profile_name> --list-details
```

  - To show only a specific tool:

```
./mcpgen client <profile_name> --list-details --tool getInventoryRecords
```

  The following output displays:

```
     Tools: 13 |    Prompts: 2

================================================================================
   TOOL DETAILS
================================================================================

________________________________________________________________________________
   Tool #1: getInventoryRecords
________________________________________________________________________________

   Description: Retrieve standard inventory stock levels and availability
   ... +8 more lines (see Complete Metadata below)

     Tags: group:inventory-management

   Input Parameters (3 total):
   • filter (string) - optional
     Flexible filter parameter supporting Progress ABL filter expr...
   • skip (integer) - optional
     Records to skip for pagination
   • top (integer) - optional
     Maximum records to return (default 100)

   Output Schema: object

   Complete Metadata (JSON):
{
  "name": "getInventoryRecords",
  "title": null,
  "description": "Retrieve standard inventory stock levels and availability\n\nContext:
 ## Filtering & Search...",
  "inputSchema": {
    "type": "object",
    "properties": {
      "filter": {
        "type": "string",
        "description": "Flexible filter parameter supporting Progress ABL filter
expressions",
        "maxLength": 500
      },
      "skip": {
        "type": "integer",
        "description": "Records to skip for pagination",
        "default": 0,
        "minimum": 0
      },
      "top": {
        "type": "integer",
        "description": "Maximum records to return (default 100)",
        "default": 100,
        "minimum": 1,
        "maximum": 1000
      }
    },
    "required": []
  },
  "outputSchema": {
    "type": "object",
    "properties": { ... }
  },
  "icons": null,
  "annotations": null,
  "meta": {
    "_fastmcp": {
      "tags": ["Inventory Management"]
    }
  }
}
```

The following table summarizes the information displayed for each tool when you run `mcpgen client <profile_name> --list-details`.

| Element | Description |
|---|---|
| Header | Specifies tool number and name with visual separator |
| Description | Specifies smart truncation for long descriptions |
| Tags | Specifies functional grouping and categorization |
| Input parameters | Specifies summary of the first five parameters |
| Output schema | Specifies type of response object |
| Examples | Specifies count if examples are available |
| Deprecation | Specifies warning if the tool is deprecated |
| Complete JSON | Specifies full metadata object for programmatic use |

## Use cases

Use the following scenarios to understand when and how to leverage the `--list-details` and related options in the `mcpgen client` command:

- **API discovery**—Quickly identify what each tool does and review the parameters it accepts.
- **Documentation generation**—Generate comprehensive documentation by parsing the JSON metadata provided for each tool.
- **Code generation**—Use the input and output schema to automatically generate client-side code.
- **Debugging**—Inspect exact parameter types, constraints, and default values for troubleshooting.
- **Single tool focus**—Apply the `--tool` option to view details for one specific tool without scrolling through all available tools.
- **Cross-platform compatibility**—Use `--args-file args.json` or `--args @args.json` syntax to avoid platform-specific issues with inline JSON.

## Use the debug option

Use the `--debug` option to display detailed diagnostic information about HTTP communication and command execution. This option is helpful for troubleshooting issues related to authentication, API responses, or network connectivity.

For example,

- This command runs the `client` operation for the specified profile and enables debug mode. Use this command when you want to review the full execution details for general client operations without invoking a specific tool.

```
./mcpgen client <profile_name> --debug
```

- This command invokes the `listProducts` tool for the specified profile, passes arguments to limit the output to three products, and enables debug mode. Use this command when you need to debug a specific tool invocation and inspect the request/response flow.

```
./mcpgen client <profile_name> --tool listProducts --args '{"limit":3}' --debug
```

Use `--debug`, when:

- Investigating authentication or authorization issues
- Troubleshooting unexpected API responses
- Verifying headers
- Debugging downstream HTTP communication
- Understanding full request/response flow

### Service account token selection

Use the `--sa-token` option to select a specific service account token.

For example:

- This command runs the `client` command for the specified profile using the `admin` service account token. Select this option when you need elevated privileges for listing tools or performing administrative operations.

```
./mcpgen client <profile_name> --sa-token admin
```

- This command lists tools for the specified profile using the `readonly` service account token. Select this option when you want to explore available tools without making changes to the environment.

```
./mcpgen client <profile_name> --sa-token readonly
```

- This command invokes the `getSupportTickets` tool for the specified profile using the `admin` token and passes arguments to limit the output to five tickets. Select this option when you need to execute a tool that requires administrative privileges and control the number of records returned.

```
./mcpgen client <profile_name> --sa-token admin
--tool getSupportTickets --args '{"limit":5}'
```

### Notes

- When listing tools with `--tool` not specified, both name and description are shown for each tool.
- `--list-json` outputs structured JSON for programmatic parsing.
- When `--json` is active and no profile is specified, the auto-profile resolution message (`[client] Using latest profile: <profile-name>`) is suppressed to ensure clean JSON output. This message continues to appear in non-JSON mode.
- `--list-details` displays comprehensive tool metadata with beautified output.
- Smart description truncation for readability.
- Complete metadata is always available in JSON section.
- Tool invocation auto-detects JSON output and pretty-prints, unless you provide `--raw`.
- File-based arguments support two syntaxes:
  - `--args @args.json` (shortcut syntax)
  - `--args-file args.json` (explicit flag)

  Both syntaxes are equivalent and read tool arguments from a JSON file instead of requiring inline JSON.
- Do not combine `--args JSON` with `--args-file args.json` or `--args @args.json` syntax.
- Exit code reflects transport/tool success; non-zero indicates failure.
- Use `--sa-token` to switch between authorization levels without changing environment variables.
- Available token names can be seen with `./mcpgen sa-tokens <profile_name>`.

## Advanced commands

These commands are intended for specialized scenarios or future enhancements. Use with caution as they may not be part of the standard workflow.

| Command | Purpose |
|---|---|
| `tool` | Internal alias for client operations, if present |
| `repair-registry` | Reconstructs the registry for consistency |
| `registry-clean` | Prunes registry noise; possible alias or extension |
| `from` | Reserved for future implementation. Derives profile from template or reference |
| `replay` | Reserved for future implementation. Re-runs captured operations sequence |

# Quick start commands to create development profiles

You can use `mcpgen` to for quickly setting up and managing development profiles with minmal configuration. Use these quick-start commands to start fast, enforce security options, update configurations, and manage environments efficiently:

| Goal | Command |
|---|---|
| Quickly start a development profile with defaults | `./mcpgen up --quick --spec <openapi.yml>` |
| Quickly start development without loading prompts | `./mcpgen up --quick --no-prompt --spec <openapi.yml>` |
| Quickly start development with custom prompts | `./mcpgen up --quick --spec <openapi.yml> --prompts ./my-prompts` |
| Quickly start development with prompts and context | `./mcpgen up --quick --spec <openapi.yml> --prompts ./prompts --context ./context` |
| Enable TLS for secure server communication | `./mcpgen up --quick --tls` |
| Enable mTLS for client and server authentication | `./mcpgen up --quick --mtls` |
| Enable TLS and include custom domain names in certificates | `./mcpgen up --quick --tls --extra-hostnames api.example.com,staging.example.com` |
| Enable mTLS and add extra hostnames | `./mcpgen up --quick --mtls --extra-hostnames localhost,127.0.0.1,myapp.local` |
| Skip service account setup (user-only authentication) | `./mcpgen up --quick --no-sa` |
| Run guided interactive setup for profile creation | `./mcpgen up --interactive` |
| Display all profiles with detailed information | `./mcpgen list --wide` |
| Export a hardened production-ready bundle | `./mcpgen export myprof --bundle prod` |
| Update prompts for a profile | `./mcpgen update myprof --prompts ./prompts` |
| Update specification and base URL | `./mcpgen update myprof --spec ./new-spec.yml --base-url https://api.example.com` |
| Reset environment | `./mcpgen terminate --force` |

# Typical workflow and API discovery

## Typical workflow

The following steps outline a standard workflow for generating profiles, exploring APIs, and deploying MCP configurations. Each step includes the command and its purpose:

1. Use the up command to quickly create a profile with TLS enabled and an OpenAPI specification:

```
./mcpgen up --quick --tls --spec <openapi.yml>
```

   This command initializes a new profile using the provided OpenAPI specification.

2. Run the `list` command to verify that the container is active:

```
./mcpgen list --wide
```

   This command displays all running containers with extended details.

3. To list available tools with names and descriptions, run:

```
./mcpgen client myprof
```

   This output provides a quick overview of all tools in the profile.

   For full metadata of all tools:

```
./mcpgen client myprof --list-details
```

   This command retrieves detailed schema information for every tool.

4. To view complete details for a single tool, for example `getInventoryRecords`, run:

```
./mcpgen client myprof --list-details --tool getInventoryRecords
```

   This command focuses on one tool and displays its schema and parameters.

5. To execute a tool with arguments:

```
./mcpgen client myprof --tool listProducts --args '{"limit":3}'
```

   This command runs the tool and returns results based on the specified arguments.

6. Optionally, create additional tokens for different access levels:

```
./mcpgen sa-token myprof --name admin --scopes service_account mcp_access mcp_admin
./mcpgen sa-token myprof --name readonly --scopes service_account mcp_access
```

   These commands generate service account tokens with defined scopes.

   To test with a specific token:

```
./mcpgen client myprof --sa-token admin
```

   This applies the selected token for the client session.

7. Enable debug mode to inspect HTTP exchanges:

```
./mcpgen client myprof --tool getOrders --args '{}' --debug
```

   This command helps troubleshooting issues by showing detailed request and response data.

8. Export the profile for production:

```
./mcpgen export myprof --bundle prod
```

   This command creates a production-ready bundle.

9. Deploy on another host using the generated Docker Compose file or helper script.

## API discovery workflow

Follow these steps to explore and test APIs:

1. See all available tools:

```
./mcpgen client myprof
```

2. Explore all tool schemas:

```
./mcpgen client myprof --list-details
```

3. View complete details:

```
./mcpgen client myprof --list-details --tool getWarehouseMemos
```

4. Invoke with sample data:

```
./mcpgen client myprof --tool getWarehouseMemos --args
'{"filter":"Warehouse_ID=\"STL01\""}'
```

5. See full HTTP exchange:

```
./mcpgen client myprof --tool getWarehouseMemos --args '{}' --debug
```

# Error patterns and remedies

This topic describes common error symptoms, their likely causes, and recommended fixes. Use these remedies to quickly resolve issues during MCP operations.

| Symptom | Likely cause | Fix |
|---|---|---|
| Specification file not found | The path is incorrect, or the volume is missing. | Re-run the `up` command with the correct `--spec` option. |
| Authentication failures (401) | The service account header is missing. | Include the `X-OEMCP-SERVICEACCOUNT` header with a valid JWT token. |
| Tools list is empty | Scopes are filtering out all the tools. | Add the required scopes or adjust `security.authorization.tools`. |
| Token not found with `--sa-token` | The specified token name does not exist. | Check available tokens using `./mcpgen sa-tokens <profile_name>`. |
| Wrong token used despite flag | An environment variable is overriding the token. | Unset `$SERVICE_ACCOUNT_JWT` or use the explicit `--sa-token` parameter. |
| Admin tools not visible | A regular token is being used. | Use `--sa-token admin` or a token with the `mcp_admin` scope. |
| Missing TLS despite config | Certificates or key files are not present. | Ensure that both the certificate file and the key file are present. For development environments, run the certificate generation script to create these files. |
| SA header double prefix | The manual prefix was added to the token file. | Allow the generator to build the header; avoid manual prefixing. |
| Empty prompt plan | `--no-prompt` was used, or no prompt files exist. | Remove the flag or add prompt `.md` files. |
| Token exchange timeout | Incorrect URL or network issue. | Validate that the exchange endpoint is reachable from the container. |

# Clean removal of profile

To remove profiles and associated resources:

- Run `./mcpgen terminate` for a full wipe, which removes profiles, Docker configurations, running containers. However, this does not uninstall the MCP Server.
- Run `./mcpgen remove` for deleting individual profiles.
- Run `./mcpgen prune` for safe cleanup of stopped profiles.

# Debug and diagnostics

The `mcpgen` CLI suppresses raw Python stack traces for end users. Errors appear as concise, formatted lines. Use these controls for deeper troubleshooting.

## Default silent mode

The `mcpgen` CLI displays error messages when operating in silent mode by default. Silent mode means that raw Python stack traces are suppressed, and errors are shown as concise, user-friendly messages instead of verbose internal details. The following outlines the default error output style for different failure scenarios in `mcpgen`:

| Situation | Output style |
|---|---|
| Unexpected internal exception | The CLI displays: `Error: Unexpected error: <message>` without showing a traceback. |
| Import or initialization failure | The CLI outputs a single-line error message without exposing Python internals. |
| Unsupported Python version (<3.10) | The CLI shows a version guard message instead of a `TypeError`, indicating that the interpreter is too old. |

## Python version guard

The `mcpgen` CLI enforces a minimum Python version requirement to ensure compatibility with modern language features and libraries. The tool requires **Python 3.10 or higher**. If you attempt to run `mcpgen` on an older interpreter, the CLI will not proceed and will display a clear error message. For example, when an older Python version is detected, the CLI emits the following message:

```
Error: Unsupported Python version. Detected 3.9. Required >= 3.10
```

This message indicates that the current environment does not meet the minimum version requirement.

To fix this issue:

1. Check your Python version:

```
python --version
```

2. Upgrade Python to 3.10 or later using your system package manager or official installer.
3. Validate your environment after upgrading:

```
./mcpgen doctor
```

   This command confirms that the dependencies meet the required specifications.

## Common troubleshooting scenarios

The following table summarizes how to quickly resolve common issues encountered during MCP operations by applying the recommended actions for each scenario:

| Scenario | Action |
|---|---|
| Suspect stale wheel code | Set the environment variable `MCPGEN_FORCE_REFRESH=1` and re-run the desired `mcpgen` command to force a clean wheel extraction. |
| Validate environment and versions | Run `./mcpgen doctor` to check Python version, dependencies, and configuration integrity. |
| Verify generated configuration quickly | Use `cat generated/<profile_name>/server/mcp_server_config.json` to inspect the generated server configuration file. |
| Images not found after unpacking | Execute `./load_images.sh` followed by `./mcpgen list` to reload images and confirm availability. |

## Minimal bug report template

Include the following when filing an issue:

- Full output from running the `./mcpgen doctor` command to help verify environment health and configuration.
- The exact command you executed, including all flags and parameters, so the issue can be reproduced accurately.
- Complete error message displayed by the CLI to assist in diagnosing the problem.
- Python version installed on your system by running `python --version`.
- The operating system (OS) and hardware architecture, for example, `Ubuntu 22.04 on x86_64` to provide context for environment-specific issues.

# 9

# Operational guidelines and performance best practices for MCP servers

This section provides guidelines for daily operations, observability, performance tuning, upgrade strategies, and security enhancements. It also includes incident response steps and runbooks for common tasks.

For details, see the following topics:


- Routine operational tasks
- Improve system observability
- Capacity and performance
- Safe upgrade process and rollback preparation
- Configuration drift control
- Prompts lifecycle
- Security posture enhancements
- Incident response snapshot
- Operational runbooks for MCP server
- Estimate key performance indicators

# Routine operational tasks

Use the following commands for routine operational tasks:

| Task | Command | Notes |
|---|---|---|
| Status snapshot | `./mcpgen list --wide` | Provides a quick inventory |
| Tail logs | `./mcpgen logs <prof> -f` | Use `grep` for tool IDs |
| Invoke health tool (if present) | `./mcpgen client <prof> --tool health` | Alternatively, use a simple `GET` tool |
| Rotate SA token (planned) | `./mcpgen remint-all <prof>` | Update clients with new JWT |

# Improve system observability

System observability is the ability to understand the internal state of a system by examining its outputs, such as logs, metrics, and traces. It goes beyond simple monitoring by providing insights into *why* something is happening, not just *what* is happening.

Follow these practices to improve system observability:

- Use structured logging in production for easier indexing.
- Leverage correlation IDs to trace issues across downstream services.
- Keep payload logging off unless investigating serialization bugs. Revert after debugging.

# Capacity and performance

This section explains how to plan for system capacity and optimize performance to ensure reliable operations under varying workloads. It covers vertical scaling limits, horizontal scaling triggers, and strategies for resource management. Understanding these guidelines helps maintain system stability and support scalability for high availability.

## Response and resource protection

The response guard is a critical mechanism that prevents excessive memory usage caused by large payloads. It ensures that the system remains responsive and avoids crashes during high-volume operations. When tools consistently approach guard limits, it is a signal that the underlying API needs a redesign to use pagination for better efficiency. Latency hotspots often occur at token exchange endpoints, during downstream TLS handshakes, and when retrieving JWKS keys at cache refresh intervals.

## Scaling architecture and performance tuning

The MCP server uses asyncio concurrency model, which is an asynchronous I/O model to achieve high throughput and low latency. This model is designed for I/O-bound workloads and enables the system to manage thousands of concurrent connections without blocking operations. Some of the key characterstics of this architecture are:

- The server uses single-threaded asynchronous I/O within each worker process.
- It employs an event-loop-based concurrency model to handle thousands of simultaneous connections.
- Non-blocking operations are used for HTTP requests and downstream API calls, which improves responsiveness.
- This architecture is highly efficient for I/O-bound workloads, which are typical in MCP server operations.

## Production scaling strategy

You can configure worker processes for different environments to achieve optimal performance and reliability. Worker configuration depends on workload characteristics such as I/O intensity, CPU requirements, and high-availability needs.

The following table and examples describe recommended worker configurations for development, production, and high-availability scenarios.

| Environment | Configuration | Purpose |
|---|---|---|
| Development | `export UVICORN_WORKERS=1` | Use a single worker for simplicity during development |
| Production (I/O-heavy) | `export UVICORN_WORKERS=2` | Recommended for workloads dominated by network I/O |
| Production (CPU-intensive) | `export UVICORN_WORKERS=$(nproc)` | Match the number of workers to available CPU cores |
| High-availability | `export UVICORN_WORKERS=$(($(nproc) * 2))` | Over-provision workers for redundancy and failover monitoring |

**Explanation**:

- In development, a single worker is sufficient because the workload is minimal and simplicity is preferred.
- For I/O-heavy production environments, two workers are recommended to handle concurrent requests efficiently without overloading the system.
- In CPU-intensive production environments, set the number of workers equal to the number of CPU cores to maximize parallel processing.
- For high-availability scenarios, double the number of workers compared to CPU cores. This approach provides redundancy and supports failover, but it requires active monitoring to avoid resource contention.

### Capacity planning

You must plan for system capacity to ensure optimal performance and scalability. System capacity is the limit of vertical scaling and it identifies triggers for horizontal scaling. Understanding these factors helps maintain stability and prevent resource exhaustion during peak loads.

Vertical scaling refers to increasing resources within a single server instance. The following limits apply:

- Memory per worker is approximately 100–200 MB as a baseline plus additional request buffers.
- CPU efficiency decreases with diminishing returns beyond twice the number of CPU cores.
- Connection limits are approximately 1000 concurrent connections per worker when using asyncio.

Horizontal scaling involves adding more server instances to distribute load. Consider scaling horizontally when:

- CPU utilization remains above 70 percent after tuning the worker count.
- Memory pressure begins to affect response times.
- Network I/O saturation occurs, which is rare for typical MCP loads.
- There is a need for geographic distribution or high availability.

## Safe upgrade process and rollback preparation

You can safely upgrade the MCP server while minimizing risk by preparing rollback options, applying the new distribution, and validating the upgrade through regression checks.

The following steps describe how to perform a controlled upgrade and ensure system stability:

1. To provide a rollback option in case of failure, export the current working profile using `git add` or create an archive.
2. Retrieve the latest version of the software by pulling the new distribution ZIP.
3. To update container images for the new release, load images using `load_images.sh`.
4. Recreate the profile with the same specification and configuration to ensure consistency between old and new setups.
5. To detect configuration differences before deployment, compare `mcp_server_config.generated.json` with the previous version.
6. Run regression tool invocations and compare representative outputs for validating functional integrity after upgrade.

## Configuration drift control

Configuration drift occurs when production settings deviate from the approved configuration baseline.To maintain consistency and prevent unauthorized changes across environments:

- Track the canonical `mcp_server_config.json` file in version control to maintain a single source of truth.
- Do not allow manual edits on production hosts. All changes must be applied through a pull request and followed by a redeployment process.

## Prompts lifecycle

Prompts are important assets that influence system behavior and user interactions. To manage prompts securely, track changes, and enforce access controls to prevent data exposure and unauthorized use:

- Treat prompts as versioned resources and review differences carefully before merging. Pay special attention to size changes and potential content risks such as exposure of personally identifiable information.
- Add appropriate scopes, such as `prompts.read`, to prompt selection tools to prevent unauthorized access.

## Security posture enhancements

Maintaining a strong security posture is essential for protecting MCP deployments against unauthorized access and data compromise.

The following table outlines key enhancements that reinforce trust, minimize attack surfaces, and ensure compliance with enterprise security principles:

| Enhancement | Value |
|---|---|
| Enable mTLS for server transport | Provides mutual trust and verifies client identity during communication. |
| Use JWKS with pinned algorithms | Supports key rotation and reduces the blast radius in case of key compromise. |
| Apply strict route maps | Minimizes unintended write operations by restricting exposed endpoints. |
| Implement scope granularity | Enforces the principle of least privilege by limiting access to necessary actions only. |
| Rotate service account keys periodically | Reduces the exposure window and mitigates risks associated with long-lived credentials. |

## Incident response snapshot

Effective incident response ensures quick recovery and minimizes risk during unexpected events.

The following table outlines common scenarios and recommended actions for MCP environments:

| Scenario | Action |
|---|---|
| Suspected service account key leak | Immediately revoke the old key to stop accepting it, generate a new key, and rotate all associated tokens. |
| Unexpected 5xx error bursts | Inspect system logs using correlation IDs and verify the availability of downstream services to identify root causes. |
| Tool abuse through high-frequency requests | Enable rate limiting and adjust the `max_requests_per_second` parameter to prevent misuse and maintain system stability. |

# Operational runbooks for MCP server

The runbooks help administrators quickly adjust configuration parameters, extend tool scopes, and enable structured logging for better observability, all while maintaining system stability and security. This topic provides concise, actionable steps for common operational tasks in the OpenEdge MCP Server environment.

The following runbooks outline quick operational steps for common MCP Server tasks:

1. **Temporarily increase array limit**:
   - Edit `runtime.http.response_guard.max_array_items`.
   - Restart the container.
   - Monitor memory usage and revert after investigation.
2. **Add a new tool scope**:
   - Append the scope name to `security.authorization.tools` mapping.
   - Issue tokens that include the new scope.
   - Validate that discovery filtering behaves as intended.
3. **Enable structured logs**:
   - Set `observability.logging.json=true`.
   - Optionally reduce verbosity by using `observability.logging.level=INFO`.
   - Restart the server and confirm JSON log shape is consistent for indexing.

# Estimate key performance indicators

Although native metrics export is not yet available, you can still approximate critical key performance indicators (KPIs) by analyzing server logs. These manual methods provide insight into tool usage, error trends, and performance characteristics until built-in observability features are introduced. Some of the manual methods include:

- **Tool invocation rate**—Count log lines containing the `Invoking tool` pattern.
- **Error ratio**—Compare the number of `4xx` and `5xx` status lines against total tool invocations.
- **Latency**—If timing middleware is enabled, parse timing records from logs to estimate response times.

# 10

# Configuration reference

The OpenEdge MCP server uses a configuration schema to define all available settings in a structured way. A configuration schema acts as a blueprint for the configuration file. It organizes properties into logical groups, specifies whether they are required or optional, and explains how they interact. This ensures consistency and makes it easier to configure the server correctly. This section explains the purpose of each group, provides examples, and highlights important notes for correct usage.

For details, see the following topics:

- Configuration groups
- Configure environment variable overrides
- Minimal configuration examples
- Guidance and common pitfalls

## Configuration groups

The configuration file for an MCP server typically includes several important groups that define the behavior and capabilities of the server. Each group focuses on a specific area of functionality.

The following table introduces the major configuration groups in the OpenEdge MCP server configuration file:

| Group | Focus |
|---|---|
| `api` | OpenAPI specification source, routing, and tagging enrichment |
| `security` | Authentication (service account and user) and authorization (scopes) |

| Group | Focus |
|---|---|
| `runtime` | Server transport, downstream HTTP, middleware, TLS and mTLS settings |
| `assistant` | Prompt directory and future assistant features |
| `observability` | Logging levels, structured mode, and per-logger overrides |

## The api group

The `api` group controls how the OpenEdge MCP server integrates with OpenAPI specifications and manages routing, tagging, and tool generation. This topic explains the subgroups of the `api` group, their purpose, and how they influence API exposure.

### OpenAPI specification settings

The `api.openapi` configuration specifies how the MCP server locates and interprets the OpenAPI specification. Correct configuration ensures that the server can load the specification and generate tools accurately.

The following table lists the properties available in the `api.openapi` subgroup and describes their purpose:

| Property | Required | Purpose | Notes |
|---|---|---|---|
| `spec_file` | Yes | Path to the OpenAPI specification | Can be relative or absolute. Resolved at server startup. |
| `base_url` | No | Default downstream base URL | Prefer `runtime.http.base_url` in new and generated configurations. `api.openapi.base_url` is still supported for compatibility. |
| `spec_name` | No | Human-readable label | Used in logs for only cosmetic purposes. |

The following example shows how to configure the `api.openapi` subgroup in the MCP server configuration file:

```
{
    "api": {
        "openapi": {
            "spec_file": "openapi.yml"
        }
    },
    "runtime": {
        "http": {
            "base_url": "https://api.example.com"
        }
    }
}
```

## Configure routes

The `routes.maps` configuration determines which operations from the OpenAPI specification are exposed as tools. If this section is omitted, non-GET methods are excluded by default, resulting in read-only behavior. This subgroup control exposure of API operations through inclusion and exclusion rules.

The `routes.maps` configuration contains several subsections that define how operations are filtered and tagged. Each subsection serves a specific purpose in controlling API exposure:

- `maps`—Orders `INCLUDE` or `EXCLUDE` rules for paths and methods.
- `entity_rules`—Attaches `entity:*` tags based on path patterns for better catalog grouping.
- `tags`—Injects additional tags for operations by path and optional method list.
- `exclude_tool_name_patterns`—Applies `fnmatch` patterns after OpenAPI load to remove generated tools by name. Useful for removing duplicate management endpoints or experimental operations without altering the upstream specification.

The system evaluates rules in the order they appear. The first matching rule is applied. It is recommended that you keep the rule set concise and provide explicit `INCLUDE` rules for any write operations that you want to expose intentionally.

## Processing pipeline

The system processes operations in three stages:

1. Route maps apply early filtering based on path, method, and tags.
2. Tools are generated from the remaining operations.
3. Pattern filtering removes tools by name using the `exclude_tool_name_patterns` property.

## Tag processing in route maps

The MCP server automatically generates normalized tags from the OpenAPI specification. For example, "APSV Transport" becomes "group:apsv-transport". It is recommended to use these generated tag names in route map filters, and not the original OpenAPI tag names. Normalization ensures consistency by converting original OpenAPI tags into a predictable format. The following examples show how to configure route maps using normalized tags and inclusion or exclusion rules.

- Allow two POST endpoints while blocking others:

```
{
  "api": {
    "routes": {
      "maps": [
        { "methods": ["POST"], "path": "/orders/**", "mcp_type": "INCLUDE" },
        { "methods": ["POST"], "mcp_type": "EXCLUDE" }
      ],
      "entity_rules": [ { "entity": "order", "path_prefix": "/orders" } ]
    }
  }
}
```

- Exclude operations by auto-generated tags:

```
{
  "api": {
    "routes": {
      "maps": [
        {
          "comment": "Exclude SOAP operations",
          "tags": ["group:soap-transport"],
          "mcp_type": "EXCLUDE"
        },
        {
          "comment": "Allow all other operations",
          "methods": ["GET","POST","PUT","DELETE","PATCH"],
          "mcp_type": "TOOL"
        }
      ]
    }
  }
}
```

## OperationId deduplication

The server automatically detects and resolves duplicate `operationId` values in OpenAPI specs before tool generation. When the system finds duplicates, it renames them by prefixing normalized tag names. For example:

```
getMetrics + getMetrics is changed to getMetrics + soap-transport-getMetrics
```

This approach ensures unique tool names without overwriting. No configuration is required for this process.

## Tool name filtering

Some OpenAPI generators emit unwanted operations or leave residual duplicates. The server provides post-generation filtering using `exclude_tool_name_patterns`. This property accepts an array of fnmatch patterns such as `"*_1"`, `"*_2"`, or `"resetMetrics_*"`. For example:

```
{
  "api": {
    "routes": {
      "maps": [
        { "path": "/**", "mcp_type": "INCLUDE" },
        { "methods": ["DELETE"], "path":
"/oemanager/rest/oeservices/unpublishService/**", "mcp_type": "EXCLUDE" },
        { "methods": ["DELETE"], "path":
"/oemanager/soap/oeservices/unpublishService/**", "mcp_type": "EXCLUDE" }
      ],
      "exclude_tool_name_patterns": ["*_1", "*_2"]
    }
  }
}
```

The following points summarize the operational behavior and recommendations for tool name filtering:

- **Processing order**—The system applies route maps first to filter operations. After that, pattern-based filtering removes tools by name.
- **Tag conflicts**—When using both tag-based exclusions and pattern filtering, verify that patterns do not remove tools before tag exclusions can take effect.
- **Path-based exclusions**—Use `EXCLUDE` rules based on paths early in the pipeline to reduce the surface area of operations.
- **Post-registration filtering**—Name pattern filtering occurs after tool registration. Tools removed by patterns do not appear in discovery or inventory logs.
- **Preferred approach**—Deduplicate `operationId` values in the OpenAPI specification whenever possible. Pattern filtering should be treated as a temporary, non-invasive solution.
- **Pattern list size**—Keep the pattern list short. Avoid overly broad patterns such as `*Service*` because they may hide required tools.
- **Verification**—At startup, the `startup` logger (INFO level) reports the number of tools removed by name patterns using the message `Filtered tools by name patterns removed=<N>`. For detailed inventory logs, set `logging.loggers.startup=DEBUG`.

## Optional parameter validation

Optional parameter validation acts as a security guardrail for tool arguments. It helps prevent malicious input from being processed by the MCP server. This feature is particularly useful when handling untrusted user input that flows from prompts to tool arguments.

The following details describe how optional parameter validation works and when to enable it:

- **Configuration location**—This feature is configured under `runtime.tools.param_validation`. A legacy alias exists, and a unified configuration location is planned for future releases.
- **Purpose**—Use this feature only when user input is untrusted. It protects against harmful patterns such as script URLs, directory traversal, shell command chaining, and Carriage Return and Line Feed (CRLF) injection.
- **Default behavior**—Parameter validation is disabled by default to avoid unnecessary restrictions in trusted environments.

# The security group

The `security` group manages authentication and authorization for the MCP server. These settings control how credentials are validated and how access to tools is restricted based on scopes. Proper configuration ensures secure communication and prevents unauthorized access.

## Authentication

Authentication verifies the identity of the server and users. The MCP server supports a dual model:

- **Service account (SA)**—Represents the server or automation identity.
- **User credentials**—Represents end-user context using tokens or JWTs.

Either or both models can be enabled. Scope-based filtering activates only when authentication is enabled.

## Authentication headers and access information

Authentication headers and tokens ensure secure communication between the MCP client and server. Proper configuration prevents unauthorized access and enforces security policies. Using mTLS adds an extra layer of trust by validating client certificates.This topic explains how authentication headers are used in OpenEdge MCP client-server communication.

## Service account authentication

Use this when the service account feature is enabled by default.

| Header | Value | Notes |
|---|---|---|
| `X-OEMCP-SERVICEACCOUNT` | JWT | Required for service account authentication |

Default JWT is stored in `generated/<profile_name>/client/client.env` under `SERVICE_ACCOUNT_JWT=`.

## Token selection priority for mcpgen client command

Tokens are selected in the following order:

| Priority | Source | Description |
|---|---|---|
| 1 | `--sa-token NAME` | Uses specific token file `client/sa.<name>.jwt` (highest priority) |
| 2 | `$SERVICE_ACCOUNT_JWT` | Environment variable override |
| 3 | `client.env` | Default token from file |

**Note:** If required scopes are defined, ensure the JWT includes them or renew using `remint-all`.

## User token authentication

When user token authentication is enabled, the client must include the token in the request header. This token provides an additional layer of security beyond service account authentication. Include the user token in the `Authorization` header of the HTTP request. The format is:

```
Authorization: Bearer <token>
```

The prefix and header name can be configured based on your environment.

**Note:** User token is not required for simple service-account-only setups.

## mTLS requirement

Mutual TLS (mTLS) provides an additional security layer by requiring both the client and server to present valid certificates during the TLS handshake. This ensures that only trusted clients can communicate with the server. When you use the `--mtls` option, the client must supply a valid certificate recognized by the certificate authority (CA) generated by the MCP tools. The CA and related scripts are located in the `certs/` directory. If you are using the containerized utilities client with generated artifacts, mTLS configuration is handled automatically.

## Configure service account authentication

The service account configuration asserts the server identity. The `security.authentication.service_account` configuration defines how the MCP server authenticates itself as a trusted entity. This is critical for environments where requests must originate from a verified source. The service account uses JWT tokens for authentication and supports two modes:

- **Token mode**—Uses a static `Ed25519` public key for verification.
- **OAuth mode**—Uses a remote JWKS endpoint for dynamic key validation.

Service account authentication may be required for every request in production environments.

The following table lists the key properties in `security.authentication.service_account`:

| Property | Purpose | Typical use |
|---|---|---|
| `enabled` | Enable or disable service account authentication | Turn off for quick proof of concept |
| `required` | Reject requests without a valid SA JWT | Enable in production |
| `mode` | Authentication mode: `"token"` or `"oauth"` | `"token"` for static keys, `"oauth"` for JWKS or OAuth proxy |
| `header` | Incoming SA header name | Default is `X-OEMCP-SERVICEACCOUNT` |
| `public_key` | Ed25519 public key (PEM) | Used for local static verification when mode is `"token"` |
| `jwks_uri` | Remote JWKS endpoint | Supports key rotation when mode is `"oauth"` |
| `issuer` | Expected `iss` claim | Used for multi-tenant trust or external OAuth providers |
| `client_id` | OAuth client ID for proxy mode | Required for OAuth proxy to external providers |
| `client_secret` | OAuth client secret for proxy mode | Required for confidential OAuth clients |
| `required_scopes` | Scopes SA must always include | Provides baseline gating |
| `prefix` | Accepted prefix (for example, `Bearer`) | Normalizes headers |
| `sso_mode` | Enable OAuth Single Sign-On | Forces mode `"oauth"`, header `"Authorization"`, prefix `"Bearer "` |
| `bearer_methods_supported` | OAuth bearer methods supported | Default is `["header"]`; can include `["body", "query"]` |
| `resource_documentation` | URL to API documentation | Supports `RFC 9728` metadata |
| `resource_policy_uri` | URL to token processing policy | Supports `RFC 9728` compliance |
| `revocation_endpoint` | OAuth token revocation endpoint | Supports `RFC 7009` |
| `introspection_endpoint` | OAuth token introspection endpoint | Supports `RFC 7662` |
| `require_metadata_on_401` | Include resource metadata in 401 responses | Default is true; adds `resource_metadata` in `WWW-Authenticate` header |

**Note:** If both `public_key` and `jwks_uri` are present, the server attempts static `Ed25519` verification first and falls back to JWKS.

Token mode

When using the token mode (`mode: "token"`), the MCP server validates service account tokens using a static public key. These tokens are JSON Web Tokens ( JWTs) signed with a private key and verified against the configured public key. This mode does not involve dynamic key discovery or OAuth flows, making it suitable for simple deployments where pre-shared keys are acceptable. However, you must ensure that the public key is securely stored and rotated periodically.

The following example shows how to configure token mode for service account authentication:

```
{
  "service_account": {
    "enabled": true,
    "mode": "token",
    "header": "X-OEMCP-SERVICEACCOUNT",
    "prefix": "Bearer ",
    "public_key": "-----BEGIN PUBLIC KEY-----\n...\n-----END PUBLIC KEY-----",
    "issuer": "my-auth-server"
  }
}
```

OAuth mode

OAuth mode is an authentication option for service accounts where the MCP server validates tokens using JSON Web Key Set (JWKS) from an external OAuth or OpenID Connect provider. This mode is designed for enterprise deployments that require integration with identity providers and support for separate service account and user tokens.

The following example shows how to configure OAuth mode for service account authentication:

```
{
  "service_account": {
    "enabled": true,
    "mode": "oauth",
    "header": "X-OEMCP-SERVICEACCOUNT",
    "prefix": "Bearer ",
    "issuer": "https://auth.example.com/realms/myapp",
    "jwks_uri": "https://auth.example.com/realms/myapp/certs",
    "algorithms": ["RS256"],
    "required_scopes": ["service.access"]
  },
  "user_auth": {
    "enabled": true,
    "header": "Authorization",
    "prefix": "Bearer "
  }
}
```

## OAuth authentication modes

OAuth-based authentication provides flexibility for integrating MCP with external identity providers and enabling Single Sign-On. These modes allow secure token validation and forwarding to downstream APIs.

The supported OAuth modes and patterns are:

- **OAuth SSO mode**—When `sso_mode=true`, the server automatically:
  - Forces `mode="oauth"`
  - Sets the header to `Authorization`
  - Applies the prefix `Bearer`
  - Sets `user_auth.enabled: false` because the service account token is used for both authentication and authorization.
  - Exposes the OAuth RFC 9728 protected resource metadata endpoint automatically.
  - Includes the `resource_metadata` parameter in `WWW-Authenticate` headers for enhanced client discovery.

  These actions enable seamless Single Sign-On where the same OAuth token validates MCP access and is forwarded to downstream APIs. Use this mode when MCP clients already have valid OAuth tokens from external providers. The following example shows how to configure OAuth SSO mode for service account authentication:

```
{
  "service_account": {
    "enabled": true,
    "mode": "oauth",
    "header": "Authorization",
    "prefix": "Bearer ",
    "issuer": "https://auth.example.com/realms/myapp",
    "jwks_uri": "https://auth.example.com/realms/myapp/certs",
    "algorithms": ["RS256"],
    "audience": "mcp-server-api",
    "required_scopes": ["api.access"],
    "advertised_scopes": ["api.read", "api.write"],
    "sso_mode": true,
    "bearer_methods_supported": ["header"],
    "require_metadata_on_401": true,
    "resource_documentation": "https://docs.example.com/api",
    "authorization_servers": ["https://auth.example.com/realms/myapp"]
  },
  "user_auth": {
    "enabled": false
  }
}
```

- **OAuth2 proxy mode (delegated authentication)**—When `mode="oauth"` with `client_id` and `client_secret` configured, the server acts as an OAuth2 authorization server implementing Authorization Code and PKCE flow while delegating user authentication to external identity providers such as Keycloak, Azure AD, or AWS Cognito. Use this mode when MCP clients need to obtain OAuth tokens through the MCP server.
- **Individual modes:**—You can use either of these modes:
  - **SSO only**—MCP client brings an existing OAuth token, which the server validates and forwards.
  - **Proxy only**—MCP client obtains an OAuth token from the MCP server without downstream forwarding.

## OAuth2 endpoints and metadata

The OAuth2 proxy endpoints are automatically installed when proxy mode is active. The following table lists the OAuth2 proxy endpoints and their purpose:

| Proxy endpoint | Purpose |
|---|---|
| `GET /authorize` | Starts the OAuth2 authorization flow and redirects to the external provider |
| `POST /token` | Exchanges the authorization code for an access token and validates PKCE |
| `GET /oauth/callback` | Handles the callback from the external OAuth provider after user authentication |

The MCP server publishes metadata endpoints to comply with OAuth2 and resource protection standards. These endpoints help clients discover authorization servers and resource details automatically. The following table lists the discovery and metadata endpoints:

| Metadata endpoint | Standard | Exposure condition | Purpose |
|---|---|---|---|
| `/.well-known/oauth-authorization-server` | RFC 8414 | Exposed automatically when OAuth2 proxy mode is active | Provides OAuth2 server metadata, including authorization and token endpoints, supported grant types (`authorization_code`), response types (`code`), and PKCE code challenge methods (`S256`, `plain`). |
| `/.well-known/oauth-protected-resource` | RFC 9728 | Exposed only when service account authentication is enabled | Provides protected resource metadata, including authorization servers, supported scopes, bearer methods, and other compliance details for OAuth client discovery. |

When authentication fails, the MCP server provides additional metadata in the response to guide clients in obtaining valid credentials. If `require_metadata_on_401=true`, which is the default value, all 401 responses include a `WWW-Authenticate` header with the Bearer realm and a `resource_metadata` parameter pointing to the protected resource metadata endpoint. This helps clients discover the correct OAuth configuration.

## OAuth2 authentication patterns

OAuth2 integration patterns define how the server interacts with external identity providers and manages token validation and forwarding. Each pattern includes configuration examples, flow descriptions, and security considerations.

- **External OAuth flow (SSO mode only)**—Used when client applications already have OAuth integration with an enterprise identity provider. The MCP server validates the token and forwards it to downstream APIs. The following example shows how to configure external OAuth flow:

```
{
  "service_account": {
    "enabled": true,
    "mode": "oauth",
    "sso_mode": true,
    "issuer": "http://localhost:8082/realms/mcp",
    "jwks_uri": "http://localhost:8082/realms/mcp/protocol/openid-connect/certs",
    "audience": "mcp-client",
    "algorithms": ["RS256"],
    "required_scopes": ["mcp_access"]

  }
}
```

**Note:** The configuration provided in the example does not include `client_id` or `client_secret`. As a result, OAuth endpoints are not provided by the MCP server in this pattern.

These steps explain how the external OAuth flow works:

1. Client obtains OAuth token directly from Keycloak or Azure.
2. Client uses token to call MCP server tools.
3. MCP server validates token through JWKS and executes tools.
4. MCP server forwards the same token to downstream APIs.

- **MCP server OAuth flow (standard pattern)**—Simplifies client OAuth integration by allowing the MCP server to handle identity provider-specific details. The following example shows how to configure MCP server OAuth flow:

```
{
  "service_account": {
    "enabled": true,
    "mode": "oauth",
    "sso_mode": true,
    "issuer": "http://localhost:8082/realms/mcp",
    "jwks_uri": "http://localhost:8082/realms/mcp/protocol/openid-connect/certs",
    "client_id": "mcp-client",
    "client_secret": "your-client-secret-here",
    "audience": "mcp-client",
    "algorithms": ["RS256"],
    "required_scopes": ["mcp_access"],
    "advertised_scopes": ["openid", "profile", "email", "mcp_access"]
  }
}
```

These steps explain how the flow works:

1. The client initiates the OAuth process through the MCP server by calling the `/authorize` endpoint and then the `/token` endpoint.
2. The MCP server delegates user authentication to an external identity provider such as Keycloak or Azure.

3. After successful authentication, the client receives an external provider token from the MCP server.
4. The client uses this token to call MCP server tools.
5. The MCP server validates the token and executes the requested tools.
6. Finally, the MCP server forwards the same token to downstream APIs for further processing.

- **Standard OAuth2 pattern (proxy and SSO combined)**—Most deployments use both features together. This pattern works as follows:
  - Set `sso_mode` to `true` to forward the same token to downstream APIs.
  - Configure `client_id` and `client_secret` to provide OAuth endpoints for MCP clients.

  This configuration allows the MCP client to obtain an OAuth token from the MCP server, use the token for MCP calls, and forward the same token to downstream APIs.

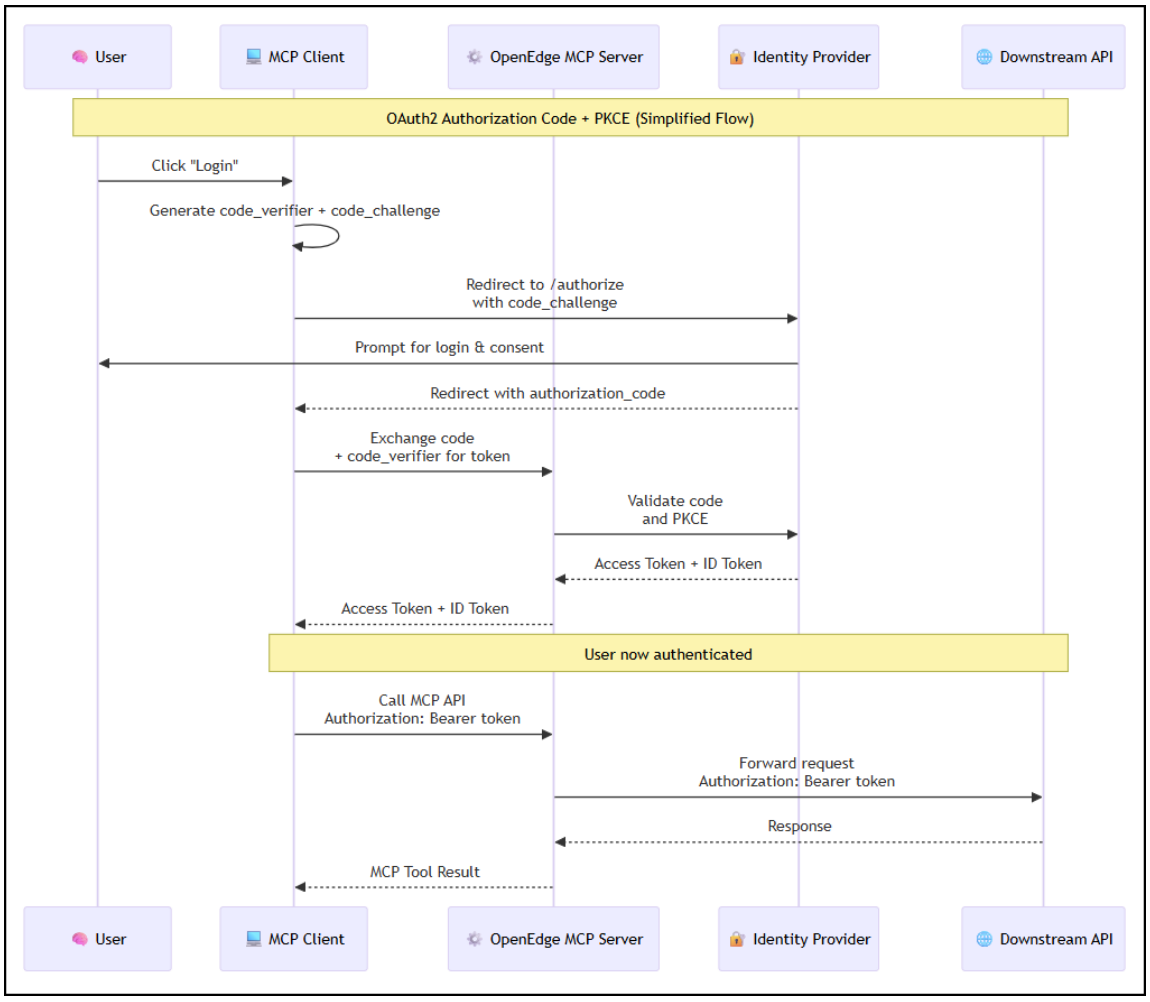
## PKCE flow for secure OAuth2 authentication

The MCP server implements the OAuth2 authorization code flow with Proof Key for Code Exchange (PKCE) to secure authentication for public clients such as browser-based MCP clients. PKCE adds cryptographic proof to prevent interception attacks by ensuring that the client initiating the authorization request is the same client exchanging the authorization code for tokens.

PKCE strengthens the OAuth2 authorization code flow by adding a code challenge and verifier mechanism. This prevents attackers from using intercepted authorization codes. Here is the PKCE security flow sequence:

1. The user initiates the process through the MCP client, which interacts with the MCP server and the identity provider.
2. During the discovery step, the client communicates with the MCP server to retrieve OAuth2 metadata from the `/.well-known/oauth-authorization-server` endpoint.
3. In the authorization step, the client sends an authorization request to the identity provider that includes a `code_challenge`.
4. The identity provider responds by returning an authorization code to the client in the callback step.
5. For token exchange, the client sends the authorization code along with the `code_verifier` to the MCP server at the `/token` endpoint.
6. For the backend exchange, the MCP server uses its `client_secret` to exchange the code with the identity provider and obtain tokens.
7. Finally, the client uses the issued token to call MCP server APIs by including the `Authorization: Bearer` header in its requests.

The sequence diagram shows the steps involved in the PKCE flow, including discovery, authorization, callback, token exchange, backend exchange, and API calls. It highlights how the MCP client and MCP server interact with the identity provider to complete the OAuth2 Authorization Code flow securely.


User
MCP Client
OpenEdge MCP Server
Identity Provider
Downstream API
OAuth2 Authorization Code + PKCE (Simplified Flow)
Click "Login"
Generate code_verifier + code_challenge
Redirect to /authorize
with code_challenge
Prompt for login & consent
Redirect with authorization_code
Exchange code
+ code_verifier for token
Validate code
and PKCE
Access Token + ID Token
Access Token + ID Token
User now authenticated
Call MCP API
Authorization: Bearer token
Forward request
Authorization: Bearer token
Response
MCP Tool Result
User
MCP Client
OpenEdge MCP Server
Identity Provider
Downstream API

The following table lists the architecture roles and their responsibilities in the PKCE flow:

| Role | Description |
|---|---|
| Identity provider (Keycloak or Azure AD) | Acts as the authentication server and verifies user credentials. |
| OpenEdge MCP server | • Acts as an OAuth2 authorization server that delegates user authentication to external identity providers<br>• Manages the OAuth2 flow<br>• Issues tokens<br>• Enforces scopes |
| MCP client | Acts as the OAuth2 client, initiates the flow, receives tokens, and makes API calls. |

The following table lists the PKCE parameters and their descriptions:

| Parameter | Description |
|---|---|
| `code_challenge` | A SHA256 hash of a random `code_verifier` sent in the authorization request. |
| `code_verifier` | A random string of 43 to 128 characters sent during token exchange. |
| `code_challenge_method` | Specifies the hashing method used for the challenge. The recommended value is `S256` for SHA256 hashing, which is more secure than `plain`. |

PKCE provides the following security benefits:

- Public client security without requiring a client secret.
- Protection against authorization code interception attacks.
- Cryptographic binding between the authorization request and token exchange.

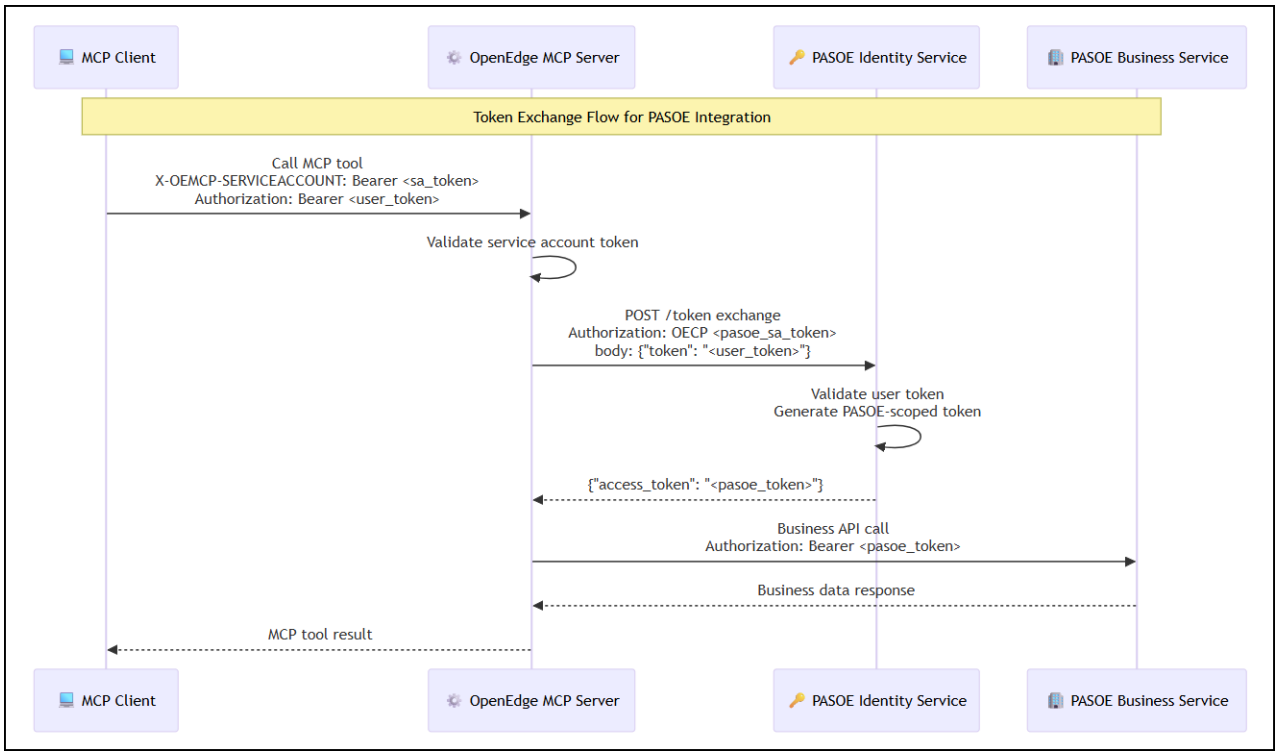
## Token exchange pattern for PAS for OpenEdge integration

The MCP server uses token exchange to integrate with Progress Application Server (PAS) for OpenEdge back-end systems. Token exchange allows the server to replace an inbound user token with a PAS for OpenEdge-scoped token that has different claims, a shorter lifetime, and limited scopes. This approach enforces the principle of least privilege and improves security for downstream API calls.

The following table describes the components involved in token exchange pattern:

| Component | Description |
|---|---|
| OpenEdge MCP server | Acts as a token exchange server. It receives the user token and exchanges it for a PAS for OpenEdge-scoped token. |
| PAS for OpenEdge SSO/Identity service | Provides an external token exchange endpoint that issues PAS for OpenEdge-compatible tokens. |

The sequence diagram shows the steps involved in token exchange pattern flow:




The token exchange process follows these steps:

1. The client sends a service account token in the `X-OEMCP-SERVICEACCOUNT` header and a user token in the `Authorization` header.
2. The MCP server validates the service account token for MCP access.
3. The MCP server extracts the user token and sends it to the PAS for OpenEdge token exchange endpoint.
4. The PAS for OpenEdge identity service validates the user token and returns a PAS for OpenEdge-scoped access token.
5. The MCP server uses the exchanged token to call PAS for OpenEdge downstream APIs.
6. The MCP server returns the tool results to the client.

The following example shows how to configure token exchange mode for PAS for OpenEdge integration:

```
{
  "service_account": {
    "enabled": true,
    "mode": "token",
    "header": "X-OEMCP-SERVICEACCOUNT",
    "public_key": "-----BEGIN PUBLIC KEY-----\n...\n-----END PUBLIC KEY-----",
    "required_scopes": ["mcp_access"]
  },
  "user_auth": {
    "enabled": true,
    "mode": "token",
    "header": "Authorization",
    "prefix": "Bearer ",
    "token_exchange": {
      "enabled": true,
      "url": "https://pasoe-server.example.com:8810/oeabl/web/identity/token",
      "method": "POST",
      "timeout_ms": 5000,
      "headers": {
        "Accept": "application/json",
        "Content-Type": "application/json",
        "Authorization": {
          "env": "TOKEN_EXCHANGE_LOGIN_TOKEN",
          "prefix": "OECP "
        }
      },
      "body": {
        "mode": "json",
        "field": "token",
        "include_prefix": false
      },
      "response": {
        "type": "json",
        "json_path": "access_token"
      }
    }
  }
}
```

Token exchange in PAS for OpenEdge provides the following security benefits:

- **Principle of least privilege**—PAS for OpenEdge receives tokens that contain only the necessary scopes and audience, ensuring minimal permissions for backend operations.
- **Reduced token lifetime**—Exchanged tokens are issued with a shorter expiration period, which limits the time window for potential misuse in backend systems.
- **Audience isolation**—PAS for OpenEdge tokens are scoped specifically for backend systems, preventing their use in unintended contexts or external services.
- **Fail-closed security**—If the token exchange process fails, the request is aborted and does not fall back to the original token, maintaining strict security controls.

For more information about key guidelines for implementing token exchange for downstream services, see Token exchange for downstream services on page 208.

Authentication flow comparison

The following table compares authentication patterns and their characteristics:

| Authentication pattern | SA token header | Downstream token | Downstream token | Token exchange | Use case |
|---|---|---|---|---|---|
| Token (Static Keys) | X-OEMCP-SERVICEACCOUNT | User token | None | Optional | Simple pre-shared keys |
| OAuth (Separate Tokens) | X-OEMCP-SERVICEACCOUNT | User token | None | Optional | Enterprise with separate SA or user tokens |
| OAuth SSO (External Flow) | `Authorization` | Same OAuth token | None | Optional | Client handles OAuth with IdP directly |
| OAuth SSO (MCP Flow) | `Authorization` | Same OAuth token | `/authorize`, `/token`, `/callback` | Optional | MCP server simplifies OAuth integration |
| Token Exchange (PAS for OpenEdge) | X-OEMCP-SERVICEACCOUNT | Exchanged PAS for OpenEdge token | None | Required | PAS for OpenEdge backend integration with scoped tokens |

**Note:** In all patterns, the MCP server performs both authentication validation and MCP tool execution. The difference is where the OAuth2 authorization flow occurs, externally or through the OAuth endpoints of the MCP server.

## Configure user authentication

The `security.authentication.user_auth` configuration adds end-user context to requests. It allows the MCP server to accept and forward user credentials to downstream services. This feature supports two modes:

- **Token mode**—Forwards an opaque token string (default).
- **JWT mode**—Accepts and validates user JWTs with optional cryptographic verification.

## Token exchange flow

Token exchange provides an additional security layer by replacing the original user token with a new token obtained from an external endpoint. The token exchange works as follows:

- When enabled, the user token is sent to an exchange endpoint using a POST request.
- The response is parsed using a JSON path expression.
- The new token is forwarded downstream using the same header and prefix configured in `user_auth`.
- Failures abort the request without falling back to the original token for deterministic security.

**Note:** The token exchange configuration is nested under `user_auth`, not at the authentication level. The runtime automatically applies `user_auth.header` and `user_auth.prefix` settings to the exchanged token.

## JWT Mode and JWKS Validation

Starting with OpenEdge MCP Server 1.1.0, JWT mode supports cryptographic validation of user tokens through JWKS (JSON Web Key Set) endpoints. This enables OAuth2/OIDC-compliant token verification, ensuring that only cryptographically valid tokens from trusted identity providers can access your MCP tools.

JWT mode operates at two validation levels:

- **Passthrough mode (default)**—Structural JWT parsing without signature verification. The server extracts claims from the JWT but does not validate the signature. This mode is suitable for development or environments where token validation is performed elsewhere. When `user_auth.mode: "jwt"` is used without `user_auth.jwks_uri`, the server performs structural validation only and does not establish trusted claims or scopes.
- **JWKS validation mode**—Full cryptographic validation using your identity provider's JWKS endpoint. When `user_auth.jwks_uri` is configured, the server validates the JWT signature using public keys from the JWKS endpoint, verifies expiration and timing claims, validates the issuer and audience, checks the algorithm against your allowlist, and extracts scopes for authorization. Tokens that fail validation are rejected with no fallback.

**When to use JWKS validation:**

Configure JWKS validation when you need authenticated user authorization with cryptographically verified tokens from identity providers such as Keycloak, Auth0, Azure AD, or Okta. Without JWKS validation, JWT mode provides structural parsing only and should not be relied upon for security-critical authorization decisions.

The following table lists the properties available in `security.authentication.user_auth`:

| Property | Purpose |
|---|---|
| `enabled` | Enables or disables user credential ingestion |
| `mode` | Specifies the mode: `token` (opaque) or `jwt` (parsed claims only) |
| `header` | Defines the incoming header name. The default value is `Authorization`. |
| `prefix` | Adds a prefix when forwarding credentials. The default value is `Bearer`. |
| `forward_downstream` | Forces forwarding even if otherwise suppressed |
| `token_exchange.*` | Configures an external `POST` request to swap the token before forwarding |
| `jwks_uri` | JWKS endpoint URL for cryptographic JWT validation. This property is required for JWKS validation mode. For example, `https://your-idp.com/.well-known/jwks.json` |
| `issuer` | Expects JWT issuer claim (`iss`) validated against the `iss` claim in incoming JWTs |
| `audience` | Expects JWT audience claim (`aud`) validated against the `aud` claim in incoming JWTs |
| `algorithms` | Algorithms allowlist for JWT signature verification. Only NIST-approved RSA algorithms are supported. The default values are ["RS256","RS384","RS512"]. Tokens with any other `alg` header are rejected. |
| `jwks_cache_seconds` | Cache lifetime for JWKS keys in seconds, which controls how long public keys are cached. The default value is 300 seconds. |

## JWKS validation configuration

When `user_auth.jwks_uri` is configured, the MCP server validates incoming user JWTs against public keys from your identity provider's JWKS endpoint. This provides cryptographic assurance that tokens are issued by a trusted authority and have not been tampered with.

**Configuration example:**

```
{
"security": {
 "authentication": {
  "enabled": true,
  "user_auth": {
   "enabled": true,
   "mode": "jwt",
   "header": "Authorization",
   "prefix": "Bearer ",
   "jwks_uri": ""https://auth.example.com/realms/my-realm/protocol/openid-connect/certs",

   "issuer": "https://auth.example.com/realms/my-realm",
   "audience": "mcp-api",
   "algorithms": ["RS256"],
   "jwks_cache_seconds": 3600
   }
  }
 }
}
```

When `jwks_uri` is configured (`mode: "jwt"`), on each user token request:

1. The JWT header kid is extracted.
2. JWKS is fetched from jwks_uri and cached for the configured cache interval.
3. The JWK matching kid is selected and converted to an RSA public key.
4. JWT verification enforces signature validity, validates exp and nbf claims, applies the algorithm allowlist, and validates iss and aud claims when issuer and audience are configured.
5. On any verification failure, the server returns hard HTTP 401; tool handlers are not reached.
6. On success, scopes are extracted from the token and used for tool gating.

If JWT validation fails for any reason, including use of `alg:none`, the server returns `HTTP 401 Unauthorized` and does not invoke tool handlers. When JWKS validation is enabled, there is no fallback to passthrough mode.

**JWKS cache behavior:**

Public keys from the JWKS endpoint are cached for the duration specified by `jwks_cache_seconds`. This reduces the number of requests to your identity provider while maintaining security. The cache is automatically refreshed when keys expire or when a JWT references a key ID (kid) that is not in the cache.

**Algorithms allowlist:**

The algorithms configuration restricts which JWT signing algorithms are accepted. For JWKS validation, only NIST-approved RSA algorithms are supported:

- RS256 (RSA with SHA-256)
- RS384 (RSA with SHA-384)
- RS512 (RSA with SHA-512)

Asymmetric algorithms such as HS256 (HMAC) are not supported in JWKS validation mode because they require shared secrets rather than public keys.

---

**Note:** Existing deployments using `user_auth.mode: "jwt"` without `user_auth.jwks_uri` continue to operate in passthrough mode. No configuration changes are required for existing deployments unless you opt for JWKS validation. To enable JWKS validation, add the `jwks_uri` configuration along with issuer, audience, and algorithms settings appropriate for your identity provider.

---

## Authentication enablement levels

Authentication behavior is controlled by three distinct switches. These switches determine whether authentication middleware, scope gating, and discovery filtering are active.

The following table lists the enablement levels, their configuration paths, and their purpose:

| Level | Key path | Scope | Purpose | Typical use |
|---|---|---|---|---|
| Global | [illegible] | Top-level | Master gate that enables or disables all authentication features. When false, both service account and user authentication settings are ignored. | Temporarily disable all authentication for debugging |
| Service account | [illegible] | Sub-section | Enables server credential injection using JWT or JWKS. Provides stable scopes for tool gating and discovery filtering. | Backend identity and consistent authorization context |
| User authentication | [illegible] | Sub-section | Accepts end-user credentials and optionally performs token exchange before forwarding downstream. | Per-request user context and least-privilege exchange |

The interaction rules define how the enablement levels interact with each other. The following points summarize the interaction rules:

- Global must be set to `true` for either sub-section to take effect.
- You can enable either sub-section independently. Combined mode forwards both credentials unless suppressed.
- Scope gating and discovery filtering activate only when global is `true` and at least one enabled credential provides scopes.
- Token exchange runs only when user authentication is enabled and `token_exchange.enabled=true`.

### Authorization

Authorization controls which tools a client can access based on assigned scopes. The MCP server uses scopes to enforce security at two levels: tool discovery and tool invocation. Proper configuration ensures that only authorized clients can view and execute specific tools.

The `security.authorization.tools` property maps tool names (or the keyword `default`) to arrays of scopes. These scopes determine whether a tool is visible and executable for a client.

### Effects of scope mapping:

Here are some of the effects of scope mapping:

- **Discovery filtering**—If a client lacks the required scopes, the tool is hidden from discovery.
- **Invocation enforcement**—If a client attempts to call a tool without the required scopes, the server returns an `insufficient_scope` error (HTTP 401) along with metadata that directs the client to obtain the correct scopes.

The following example shows how to configure scope mappings for tools:

```
{
  "security": {
    "authorization": {
      "tools": {
        "default": ["mcp_access"],
        "select_prompts": ["prompts.read"],
        "list_all_tags": ["schema.read"],
        "get_tool_schema": ["schema.read"],
        "get_filter_fields": ["schema.read"]
      }
    }
  }
```

### Best practice

Always define a `default` scope to provide consistent baseline gating for all tools, which ensures that unauthorized clients cannot access tools without explicit permission.

## The runtime group

The `runtime` group defines operational settings that control how the server listens for requests, interacts with downstream services, and applies guardrails or middleware. These settings form the execution environment plumbing, which includes transports, timeouts, TLS configurations, and resource limits. They do not represent business policies but rather technical safeguards and defaults. The `runtime` group ensures safe and predictable execution by applying defaults that favor stability. For example, responses are bounded, timing middleware is disabled unless explicitly enabled, and HTTP client settings can inherit trust stores for secure communication.

### Configure server transport settings

The server transport settings define how the runtime server accepts incoming requests and manages communication protocols. These settings control the transport type, network binding, listening port, request paths, and TLS security options. Proper configuration ensures secure and reliable connectivity for client applications.

The following table lists the key properties for the `runtime.server.transport` configuration, along with their default values and purpose:

| Property | Default | Purpose |
|---|---|---|
| `type` | `streamable-http` | Specifies the transport type. Supported values are `streamable-http` or `sse`. |
| `host` | `0.0.0.0` | Defines the network interface to bind the server. |
| `port` | `8500` | Sets the port on which the server listens for incoming requests. |
| `path` | `/http` | Indicates the request path. Use `/sse` when Server-Sent Events (SSE) are enabled. |
| `tls.certfile / keyfile` | `unset` | Enables HTTPS when both certificate and key files are provided. |
| `tls.client_cafile` | `unset` | Specifies the CA bundle for client certificates when using mutual TLS (mTLS). |
| (activation) | — | HTTPS is activated only if both certificate and key files exist. |
| `tls.require_client_cert` | `false` | Enforces client certificate validation when set to `true`. |

## Configure TLS and mTLS certificates

The OpenEdge MCP Server supports flexible certificate configuration for both server-side TLS and mTLS authentication. The following table lists the supported file formats and their purpose. This topic explains how to configure TLS and mutual TLS (mTLS) certificates for the OpenEdge MCP Server.

## Certificate file requirements

The OpenEdge MCP Server supports flexible certificate configuration for both server-side TLS and mTLS authentication. The following table lists the supported file formats and their purpose:

| Format | Purpose |
|---|---|
| `.pem` | Preferred format for certificates |
| `.crt` | Alternative format for certificates |
| `.key` | Used for private keys |

TLS requires specific files to enable secure communication. The following table lists these required files for TLS:

| File | Purpose |
|---|---|
| `certfile` | Server certificate containing public key and certificate chain |
| `keyfile` | Server private key |

mTLS adds the following additional files for client certificate validation:

| File | Purpose |
|---|---|
| `client_cafile` | Client Certificate Authority bundle for validating client certificates (mTLS only) |
| `require_client_cert` | Must be set to `true` to enforce client certificate validation |

## File naming conventions

You can use any filenames for certificates because the server reads certificate paths from configuration. The following table shows default names generated when using `mcpgen` or development certificate generation:

| Certificate type | File name |
|---|---|
| Server certificate | `localhost.pem` |
| Server private key | `localhost-key.pem` |
| Alternative server cert | `localhost.crt` |
| Client CA bundle | `client-ca.pem` |
| Client certificate | `client.pem` |
| Client private key | `client.key` |
| Combined client cert and key | `client-combined.pem` |
| Alternative client certificate | `client.crt` |

## Configuration examples

The following examples demonstrate how to configure TLS and mTLS in the runtime configuration.

Add the relevant configuration to `mcp_server_config.json`.

**Note:** After updating `mcp_server_config.json`, restart the MCP server for the changes to take effect:

```
mcpgen restart <container-name>
```

- This example shows the basic TLS configuration (server-only):

```
{
  "runtime": {
    "server": {
      "transport": {
        "tls": {
          "certfile": "./certs/my-server.pem",
          "keyfile": "./certs/my-server-key.pem"
        }
      }
    }
  }
}
```

- This example shows the mTLS configuration:

```
{
  "runtime": {
    "server": {
      "transport": {
        "tls": {
          "certfile": "./certs/server.crt",
          "keyfile": "./certs/server.key",
          "client_cafile": "./certs/trusted-clients-ca.pem",
          "require_client_cert": true
        }
      }
    }
  }
}
```

- This example showshow to use custom certificate names:

```
{
  "runtime": {
    "server": {
      "transport": {
        "tls": {
          "certfile": "./ssl/company-api-server.crt",
          "keyfile": "./ssl/company-api-private.key",
          "keyfile_password": "optional-password-for-encrypted-key"
        }
      }
    }
  }
}
```

The following examples show the `mcp_server_config.json` file structure with TLS and mTLS configured. Use them as a structural reference and replace the placeholder values with those specific to your environment.

- TLS configuration

```
{
  "api": {
    "openapi": {
      "spec_file": "openapi.yml"
    }
  },
  "security": {
    "authentication": {
      "enabled": false
    }
  },
  "runtime": {
    "server": {
      "name": "openedge-mcp-server",
      "transport": {
        "type": "streamable-http",
        "host": "0.0.0.0",
        "port": 8500,
        "path": "/http",
        "tls": {
          "certfile": "./certs/dev/localhost.pem",
          "keyfile": "./certs/dev/localhost-key.pem"
        }
      }
    },
    "http": {
      "base_url": "https://api.example.com"
    }
  }
}
```

- mTLS configuration

```
{
  "api": {
    "openapi": {
      "spec_file": "openapi.yml"
    }
  },
  "security": {
    "authentication": {
      "enabled": false
    }
  },
  "runtime": {
    "server": {
      "name": "openedge-mcp-server",
      "transport": {
        "type": "streamable-http",
        "host": "0.0.0.0",
        "port": 8500,
        "path": "/http",
        "tls": {
          "certfile": "./certs/dev/localhost.pem",
          "keyfile": "./certs/dev/localhost-key.pem",
          "client_cafile": "./certs/dev/client-ca.pem",
          "require_client_cert": true
        }
      }
    },
    "http": {
      "base_url": "https://api.example.com"
    }
  }
}
```

For additional ready-to-use examples, see the `examples` directory installed with the product.

## Override configuration with environment variables

You can override TLS configuration using environment variables. The following table lists the supported variables:

| Environment variable | Configuration path | Purpose |
|---|---|---|
| `SSL_CERTFILE` | `runtime.server.transport.tls.certfile` | Server certificate file path |
| `SSL_KEYFILE` | `runtime.server.transport.tls.keyfile` | Server private key file path |
| `SSL_CLIENT_CAFILE` | `runtime.server.transport.tls.client_cafile` | Client CA certificate for mTLS |
| `SSL_REQUIRE_CLIENT_CERT` | `runtime.server.transport.tls.require_client_cert` | Set to `1` to require client certificates |
| `SSL_KEYFILE_PASSWORD` | `runtime.server.transport.tls.keyfile_password` | Password for encrypted private keys |

## Behavioral considerations for TLS and mTLS

The following behaviors apply when configuring TLS and mTLS:

- **File existence validation**— TLS activation depends on the presence of required certificate files. The following points summarize this behavior:
  - TLS is enabled only when both `certfile` and `keyfile` exist on disk.
  - If any certificate file is missing, the server falls back to HTTP without TLS.
  - This fallback prevents misleading logs that might indicate TLS is enabled when certificates are not properly mounted.
- **Path resolution**—The server resolves certificate file paths based on the following rules:
  - Relative paths are resolved from the working directory where the server starts.
  - Absolute paths are used as provided in the configuration.
  - Ensure that all certificate files have proper file permissions so they are readable by the server process.
- **Certificate validation logic**—The server performs validation checks during startup to ensure certificate integrity:
  - The server verifies that both certificate and key files are accessible.
  - TLS activation fails if the certificate and private key do not match or are invalid.
  - For mTLS, a valid `client_cafile` is required when `require_client_cert` is set to `true`.

## Security best practices

Follow these best practices to maintain security:

- **File Permissions**—Set certificate files to be readable only by the server process, for example `chmod 600`.
- **Key Protection**—Never commit private keys to version control.
- **Certificate Rotation**—Implement regular certificate rotation. For more information, see Upgrade process and key or certificate rotation on page 257.
- **Hostname Validation—**Ensure certificates include appropriate Subject Alternative Names (SANs).
- **Strong Ciphers**—Use modern TLS versions and strong cipher suites in production.

## Generate certificates for development

Use the following commands to generate development certificates:

```
# Generate development certificates with custom hostnames
./mcpgen up --tls --extra-hostnames "api.example.com,staging.example.com"

# Generate mTLS certificates (includes client certificates)
./mcpgen up --mtls --extra-hostnames "api.example.com,staging.example.com"
```

For production certificate management and rotation procedures, see Upgrade process and key or certificate rotation on page 257.

## Troubleshoot common issues

The following table lists common issues and their resolutions:

| Issue | Resolution |
|---|---|
| "TLS disabled, falling back to HTTP" | Verify that both `certfile` and `keyfile` paths exist and are readable |
| Certificate validation errors | Check that the certificate and private key match using OpenSSL tools |
| mTLS handshake failures | Ensure the client CA bundle includes the signing CA for client certificates |
| Hostname mismatches | Add required hostnames to certificate SANs or use wildcard certificates |

## Configure downstream HTTP settings

The downstream HTTP settings control how outbound HTTP requests are constructed, secured, and logged. Proper configuration ensures reliable communication with external services while maintaining security and performance.

The following table lists the properties available under `runtime.http` and their purpose:

| Property | Purpose |
|---|---|
| `base_url` | Provides a fallback base URL when the operation URL is not absolute. |
| `default_headers` | Specifies non-secret headers applied to every outbound HTTP call. |
| `accept_encoding` | Indicates compression or logging hints for downstream requests. |
| `response_guard.*` | Defines response size and array limits. See the guard section for details. |
| `tls.*` | Configures downstream trust and optional client certificate identity. |
| `debug_raw` | Enables development logging by emitting sanitized raw HTTP request and response lines. |

## Raw HTTP debug logging

The `debug_raw` property provides enhanced visibility into HTTP requests and responses for troubleshooting purposes. This feature is available globally for outbound calls and specifically for token exchange operations. The following table lists raw HTTP debug logging details:

| Location | Key path | Scope | What is logged | Header handling |
|---|---|---|---|---|
| Global outbound | `runtime.http.debug_raw` | All downstream calls | Method, URL, status, timing, selected headers, truncated body preview | Sensitive headers partially masked. Tokens and prefixes are truncated. Secrets are not fully printed. |
| Token exchange | `security.authentication.user_auth.token_exchange.debug_raw` | Token exchange POST and response only | Exchange request and response, status, JSON path extraction details | Authorization and exchange headers masked except for prefix. Body fields logged only when small. |

Follow these guidelines when enabling raw HTTP debug logging:

- Use this feature only in development or during short diagnostic windows.
- Disable it in production to reduce log noise and prevent accidental exposure of secrets.
- Bodies larger than approximately 4 KB are logged as a size summary instead of full content.
- Binary or non-UTF payloads are summarized by size in hexadecimal format.
- Masking strategy ensures that authorization-like headers show only the prefix and a few characters. Full token values are never printed.
- Custom headers containing `token`, `secret`, `key`, or `auth` are abbreviated.
- Token exchange failures redact sensitive response fields and log only structural error information.

The following example shows how to enable raw HTTP debug logging in the configuration:

```
"runtime": { "http": { "debug_raw": true } }
"security": { "authentication": { "user_auth": { "token_exchange": { "debug_raw": true
 } } } } }
```

To disable debug logging, remove these flags or set them to `false` after diagnosing the issue.

The debug logging feature provides sufficient correlation and payload insight to troubleshoot authentication, routing, and token exchange issues without exposing full secrets in logs. This approach preserves auditability while limiting credential exposure.

## Configure response guard settings

The response guard feature controls the size and structure of responses returned by downstream services. Response guard settings help prevent excessive memory usage and protect LLM pipelines from runaway payloads.

The response guard operates in different modes that determine how oversized responses are handled. The following table summarizes the available modes and their behavior:

| Mode | Behavior |
|---|---|
| `trim` | Truncates arrays to `max_array_items` and shortens large responses. This is the default mode. |
| `block` | Rejects responses that exceed configured limits and returns an error. |
| `warn` | Allows responses that exceed limits but logs a warning for visibility. |

Response guard enforces limits on array size and response payload size. These limits prevent uncontrolled growth in memory and context.

| Limit | Default value | Purpose |
|---|---|---|
| `max_array_items` | 50 | Restricts the maximum number of items in an array. |
| `max_response_bytes` | 1 MiB | Restricts the maximum size of a response payload. |

## Configure middleware settings

The middleware components in the OpenEdge MCP Server enhance reliability, security, and performance. Middleware provides features such as error handling, logging, request timing, rate limiting, and circuit breaking. Proper configuration ensures production hardening and prevents resource exhaustion.

The following table lists the `runtime.middleware` subsections, their configurable fields, default values, and production hardening recommendations:

| Subsection | Fields (flags / knobs) | Defaults | Production hardening |
|---|---|---|---|
| `error_handling` | `enabled`, `transform_errors`, `include_traceback` | `true`, `true`, `false` | Enabled by default |
| `logging` | `enabled`, `structured`, `include_payloads` | `true`, `false`, `false` | Enabled by default<br>For more information, see Configure logging and related middleware on page 91. |
| `timing` | `enabled` | `false` | Optional<br>For more information, see Configure timing middleware on page 94. |
| `rate_limiting` | `enabled`, `max_rps`, `burst`, `global` | `false`, `10`, `20`, `false` | Enable for production<br>For more information, see Rate limiting in OpenEdge MCP Server on page 251. |
| `request_timeout` | `enabled`, `timeout_ms` | `true`, `30000` | • Critical for production environments.<br>• Terminates requests that exceed the configured timeout (default 30000 ms) to prevent resource exhaustion.<br>• To change the timeout, set `runtime.middleware.request_timeout.timeout_ms` to your desired value.<br>• Enabled by default. |

| Subsection | Fields (flags / knobs) | Defaults | Production hardening |
|---|---|---|---|
| `method_validation` | `enabled` | `true` | • Critical for production environments.<br>• Rejects unknown tool names before expensive parameter parsing. Enabled by default to reduce attack surface.<br>• Enabled by default. |
| `circuit_breaker` | `enabled`, `error_threshold`, `window_seconds`, `min_requests` | `false`, `0.5`, `60`, `10` | • Critical for production environments.<br>• Monitors error rates in a sliding window and rejects requests when thresholds are exceeded to prevent cascading failures.<br>• Disabled by default, however it can be enabled explicitly for production with tuned thresholds. |

## Additional runtime hardening options

Additional runtime settings help prevent resource exhaustion:

- **`runtime.max_request_bytes`**—Rejects oversized payloads at the ASGI layer before MCP parsing. Default is 10 KB. Configure per use case and keep ≤100 KB for production.
- **`runtime.http.limits`**—Connection pool limits prevent resource exhaustion from excessive connections. Default values are:
  - `max_connections`: 100
  - `max_keepalive_connections`: 20

### Example production configuration

The following example shows a recommended production configuration for middleware and runtime hardening:

```
{
  "runtime": {
    "max_request_bytes": 10240,
    "http": {
      "limits": {
        "max_connections": 100,
        "max_keepalive_connections": 20
      }
    },
    "middleware": {
      "request_timeout": {
        "enabled": true,
        "timeout_ms": 30000
      },
      "method_validation": {
        "enabled": true
      },
      "circuit_breaker": {
        "enabled": true,
        "error_threshold": 0.5,
        "window_seconds": 60,
        "min_requests": 10
      },
      "rate_limiting": {
        "enabled": true,
        "max_requests_per_second": 10.0,
        "burst_capacity": 20
      }
    }
  }
}
```

### Validate tool parameters

Ensure that `max_total_bytes` and pattern lists match your threat model. Leave this configuration disabled unless handling multi-tenant or untrusted input. For more information, see Optional parameter validation on page 163.

## The assistant group

The assistant group manages prompt-related settings that influence how prompts are stored and accessed during runtime. While the current implementation focuses on directory scanning, future enhancements may include prompt indexing strategies and embedding cache options. The `assistant.prompts.dir` property specifies the directory scanned for Markdown prompt files. The default location is `work/server/prompts` inside the container. The assistant group includes a size guard that limits the total size of prompt content. This safeguard prevents excessive memory usage and ensures that prompts remain lightweight and efficient.

## The observability group

The the `observability` group controls logging behavior and OpenAPI enrichment visibility in the OpenEdge MCP Server. These settings determine what information is surfaced externally for monitoring and debugging purposes. They do not enforce security policies. Increasing OpenAPI enrichment visibility, such as annotation logging, can help diagnose specification issues. However, enabling it in production environments may generate excessive log noise.

Observability settings allow you to configure log verbosity, format, and enrichment options. The following table summarizes the available settings and their purpose:

| Setting | Description |
|---|---|
| `logging.level` | Sets the root log level. The default value is `INFO`. |
| `logging.json` | Enables structured JSON logs for better machine readability. |
| `logging.loggers` | Allows per-logger level overrides for fine-grained control. |
| `format` | Specifies a printf-style template for log messages. Ignored when `logging.json` is `true`. |
| `openapi_logging` or `openapi_visibility` | Toggles OpenAPI specification enrichment and annotation verbosity. For more information about detailed keys, see OpenAPI visibility and context management on page 217. |

# Configure environment variable overrides

This section explains the essential environment variables that can override configuration settings in the OpenEdge MCP Server. Environment variables are intended for last-mile operational adjustments or runtime injection. For production environments, prefer using explicit JSON configuration files under version control to ensure reproducibility.

The following table summarizes the essential environment variables and their descriptions:

| Environment variable | Description |
|---|---|
| MCP_SERVER_CONFIG | Specifies the path to the configuration JSON file, which can be absolute or relative. |
| SERVICE_ACCOUNT_PUBLIC_KEY | Provides an inline PEM for the service account public key. |
| SERVICE_ACCOUNT_PUBLIC_KEY_FILE | Specifies the path to a PEM file containing the service account public key. |
| MCP_USE_SSE | Forces SSE transport for legacy client support. |
| OPENAPI_DUMP_SPEC | Writes the enriched OpenAPI specification to a file for debugging purposes. |
| OPENAPI_LOG_ANNOTATIONS | Controls logging of enrichment annotations. |
| OPENAPI_LOG_ANNOTATIONS_VERBOSE | Enables verbose annotation logging with per-operation and tag details. |
| DOWNSTREAM_CERTS_DIR | Specifies the directory for downstream CA bundles and optional client cert/key for mTLS. |
| TLS_CERTS_DIR | Provides an alternate path for downstream trust and client TLS assets. |
| TLS_P12_PASSWORD | Password for PKCS#12 bundle if encrypted. |
| TLS_KEY_PASSWORD | Password for encrypted private key if used. |
| TOKEN_EXCHANGE_LOGIN_TOKEN | Authorization header value for token exchange, used with the configured prefix. |

**Note:** Avoid relying heavily on environment variables for production reproducibility. Use explicit JSON configuration files under version control to maintain consistency and traceability.

# Minimal configuration examples

This topic provides minimal configuration examples for common deployment scenarios. These examples demonstrate how to configure authentication, authorization, TLS, and OAuth settings for the OpenEdge MCP Server. Use these examples as starting points and adapt them to your environment.

- **Read-only mode without authentication**—The following example shows a minimal configuration for read-only mode without authentication.

```
{
  "api": {
    "openapi": {
      "spec_file": "openapi.yml"
    }
  },
  "security": {
       "authentication": {
      "enabled": false
    }
  }
}
```

- **Service account with scopes and TLS enabled**—The following example shows a configuration for service account authentication with scopes and TLS.

```
{
  "api": {
    "openapi": {
      "spec_file": "openapi.yml",
      "base_url": "https://api.example.com"
    }
  },
  "security": {
    "authentication": {
      "enabled": true,
      "service_account": {
        "enabled": true,
        "required": true,
        "header": "X-OEMCP-SERVICEACCOUNT",
        "issuer": "https://idp.example/realms/mcp",
        "jwks_uri": "https://idp.example/realms/mcp/protocol/openid-connect/certs",
        "required_scopes": [
          "mcp_access"
        ]
      },
      "user_auth": {
        "enabled": false
      }
    },
    "authorization": {
      "tools": {
        "default": [
          "mcp_access"
        ],
        "select_prompts": [
          "prompts.read"
        ]
      }
    }
  },
  "runtime": {
    "server": {
      "transport": {
        "tls": {
          "certfile": "./certs/server.pem",
          "keyfile": "./certs/server-key.pem"
        }
      }
    }
  }
}
```

**Note:** Ensure both `certfile` and `keyfile` paths exist. If they are missing, the server will remain in HTTP mode.

- **OAuth SSO mode when client has an existing OAuth token**—The following example shows a configuration for OAuth SSO mode.

```
{
  "api": {
    "openapi": {
      "spec_file": "openapi.yml",
      "base_url": "https://api.example.com"
    }
  },
  "security": {
    "authentication": {
      "enabled": true,
      "service_account": {
        "enabled": true,
        "required": true,
        "mode": "oauth",
        "sso_mode": true,
        "issuer": "http://localhost:8082/realms/mcp",
       "jwks_uri": "http://localhost:8082/realms/mcp/protocol/openid-connect/certs",

        "audience": "mcp-client",
        "algorithms": ["RS256"],
        "required_scopes": ["mcp_access"]
      }
    },
    "authorization": {
      "tools": {
        "default": ["mcp_access"],
        "select_prompts": ["prompts.read"]
      }
    }
  }
}
```

**Note:**

- The MCP client already has an OAuth token from the identity provider.
- The server validates the token and forwards the same token to downstream APIs.
- No OAuth endpoints are provided because the client handles the OAuth flow externally.

- **OAuth2 standard configuration with proxy and SSO combined**—The following example shows a configuration for OAuth2 standard mode with proxy and SSO combined.

```
{
  "api": {
    "openapi": {
      "spec_file": "openapi.yml",
      "base_url": "https://api.example.com"
    }
  },
  "security": {
    "authentication": {
      "enabled": true,
      "service_account": {
        "enabled": true,
        "required": true,
        "mode": "oauth",
        "sso_mode": true,
        "issuer": "http://localhost:8082/realms/mcp",
       "jwks_uri": "http://localhost:8082/realms/mcp/protocol/openid-connect/certs",

        "client_id": "mcp-client",
        "client_secret": "your-client-secret-here",
        "audience": "mcp-client",
        "algorithms": ["RS256"],
        "required_scopes": ["mcp_access"],
        "advertised_scopes": ["openid", "profile", "email", "mcp_access"]
      }
    },
    "authorization": {
      "tools": {
        "default": ["mcp_access"],
        "select_prompts": ["prompts.read"]
      }
    }
  }
}
```

**Note:**

- This pattern combines proxy and SSO functionality.
- The MCP server provides OAuth endpoints such as `/authorize`, `/token`, and `/oauth/callback`.
- The client obtains a Keycloak token via the MCP server OAuth flow.
- The same token is used for MCP calls and forwarded to downstream APIs.
- The server exposes `/.well-known/oauth-authorization-server` for OAuth2 discovery.

# Guidance and common pitfalls

This topic provides important guidance and highlights common pitfalls when configuring the OpenEdge MCP Server. Following these recommendations ensures reliable operation and prevents misconfigurations that can lead to failures or security gaps.

The following list summarizes common pitfalls and their recommended remedies:

- Forgetting to set `base_url` causes relative downstream URLs to fail. Always configure either `api.openapi.base_url` or `runtime.http.base_url`.
- Enabling authentication without defining scopes makes discovery work but renders access control meaningless. Define a default scope early in the configuration.
- Disabling the response guard during early development is acceptable, but re-enable it before load testing or LLM integration to prevent runaway payloads.
- Avoid large prompt directories. The aggregate size guard prevents excessive memory usage and ensures predictable performance.

**Note:** For detailed instructions on generating profiles and certificates, see Use the mcpgen CLI for profile generation and management on page 119.

# 11

# Security and authentication architecture for MCP server

Security and authentication are critical components of any enterprise-grade system. The MCP server implements multiple layers of protection, including dual authentication, scope-based access control, token rotation, and TLS/mTLS transport security. These measures safeguard sensitive data, prevent unauthorized tool invocation, and ensure interoperability with external identity providers through OAuth2 and PKCE flows. This topic provides an overview of the security and authentication mechanisms implemented in the MCP server. It explains how these mechanisms protect against unauthorized access, ensure secure communication, and maintain compliance with industry standards.

**Disclaimer:** This document provides general security guidance for remote system configurations including the MCP Server. The documented techniques must be fully vetted and evaluated by the customer for completeness and viability in their unique deployment scenarios.

For details, see the following topics:

- Threat model and mitigation strategies
- Dual authentication process
- JWKS retrieval and caching for JWT verification
- Scope gating for discovery and execution
- Protected resource metadata and RFC 9728 compliance
- Token exchange for downstream services
- TLS and mTLS for secure communication

- Response guard strategy
- Logging hygiene
- Key and token rotation cadence

# Threat model and mitigation strategies

An abbreviated threat model is a simplified version of a full threat model. Instead of detailing every possible attack vector, actor, and scenario, it focuses on the most critical risks and their corresponding mitigations. This topic explains the abbreviated threat model for the MCP server. It identifies common security risks and describes the mitigation strategies implemented to reduce their impact. Understanding these risks helps administrators and developers apply best practices for secure deployments.

The following table lists the identified risks and the corresponding mitigation strategies:

| Risk | Mitigation strategy |
|---|---|
| Unauthorized tool invocation | Dual authentication, scope gating, and route maps |
| Service account leakage to downstream | Header mapping removes service account headers |
| Oversized payload causing memory pressure | Response guard and optional parameter validation |
| Prompt exfiltration | Size guard and scope gating for prompt tools |
| Token replay without rotation | Rotation procedures and adherence to expiration claims |

Where,

- Dual authentication ensures that both service accounts and end-user credentials are validated before tool execution.
- Scope gating restricts access to tools based on predefined scopes, reducing exposure.
- Route maps specify which API routes or tool endpoints can be accessed and under what conditions.
- Header mapping prevents sensitive service account headers from being forwarded to downstream services.
- Response guard and parameter validation protect against resource exhaustion caused by oversized payloads.
- Rotation procedures enforce token freshness and prevent replay attacks by adhering to expiration claims.

# Dual authentication process

The MCP server enforces dual authentication to secure tool invocation. Dual authentication validates both the service account and the end-user identity before processing requests. This approach ensures that automation scripts and human users are authenticated independently, reducing the risk of unauthorized access.

The components of dual authentication are:

| Authentication type | Description |
|---|---|
| Service account | Uses JWT (Ed25519 static or JWKS). Provides server or automation identity. For detailed information, see Configure service account authentication on page 165. |
| User authentication | Uses opaque token or structured JWT. Conveys end-user context. For detailed information, see Configure user authentication on page 177. |

For more information on service acount and user credentials authentication, see Authentication on page 164.

For more detailed information on service account authentication modes, PKCE flow for secure OAuth2 authentication, and Token exchange pattern for PAS for OpenEdge integration, see Configure service account authentication on page 165.

### Header normalization flow

Header normalization flow is the process of sanitizing and restructuring HTTP headers during authentication so that only the necessary credentials are forwarded downstream. This prevents sensitive information, such as service account tokens, from leaking to other services.

These steps outline how headers are normalized during authentication:

1. Extract the service account header (for example, `X-OEMCP-SERVICEACCOUNT`) and verify the JWT if enabled.
2. Extract the user credential. If token exchange is enabled, exchange the credential for a new token.
3. Compose the outbound `Authorization` header using only the user token. The service account header is never forwarded downstream.

## JWKS retrieval and caching for JWT verification

When verifying JSON Web Token (JWTs) signed with asymmetric algorithms, such as `RS256`, the system needs access to the public keys of the issuer. These keys are typically published in a JSON Web Key Set (JWKS) endpoint. This topic explains how the JWKS flow works during application startup and subsequent token validations.

The JWKS flow ensures that the system retrieves and caches public keys efficiently to minimize authentication failures. The following table summarizes the process:

| Step | Description |
|---|---|
| Startup | No JWKS fetch occurs initially. Retrieval is deferred until the first JWT verification that requires `RS256` keys. |
| First JWT verification | When a JWT signed with `RS256` is encountered, the system downloads the JWKS from the configured endpoint. |
| Caching | The downloaded keys are cached for the duration specified by `jwks_cache_seconds`. |
| Cache expiry | When the cache expires, the system attempts to re-fetch the JWKS. |
| Failure handling | If the re-fetch fails, the system continues using the previously cached keys until the next successful attempt. This design minimizes authentication outages. |

# Scope gating for discovery and execution

Scope gating ensures that only authorized tools are visible and executable by clients. This mechanism reduces security risks such as catalog enumeration attacks and provides a structured way for clients to remediate authorization failures.

Scope gating applies to two key phases: Discovery and Execution. The following table summarizes the behavior in each phase:

| Phase | Description |
|---|---|
| Discovery | Filters out tools that do not match the caller's scopes. This prevents unauthorized tools from appearing in the catalog and reduces the risk of enumeration attacks. |
| Execution | If a client attempts to execute a tool without the required scope, the system returns an HTTP 401 response. The response includes a `WWW-Authenticate` header and resource metadata (if configured) to help the client remediate the issue. |

# Protected resource metadata and RFC 9728 compliance

The MCP server complies with OAuth 2.0 Protected Resource Metadata as defined in RFC 9728. This compliance enables clients to discover resource details, authorization servers, and supported token methods automatically. It also improves interoperability and security by exposing metadata endpoints and enhancing error responses.

## Automatic endpoint exposure

The MCP server automatically exposes the protected resource metadata `/.well-known/oauth-protected-resource` endpoint when `service_account.enabled=true`. This endpoint provides essential details about the resource and its associated authorization servers.

The following example shows the JSON structure returned by the metadata endpoint:

```
{
  "resource": "https://mcp-server.example.com",
  "authorization_servers": ["https://auth.example.com/realms/myapp"],
  "jwks_uri": "https://auth.example.com/realms/myapp/certs",
  "scopes_supported": ["api.read", "api.write", "admin.access"],
  "bearer_methods_supported": ["header", "body"],
  "resource_documentation": "https://docs.example.com/api/mcp-server",
  "resource_policy_uri": "https://example.com/policies/oauth",
  "revocation_endpoint": "https://auth.example.com/.../revoke",
  "introspection_endpoint": "https://auth.example.com/.../introspect"
}
```

## Enhanced 401 responses

When `require_metadata_on_401=true`, which is the default value, all **401 Unauthorized** responses include a `WWW-Authenticate` header with a link to the resource metadata. This helps clients discover the metadata and remediate authorization issues. Here is an example of the header:

```
WWW-Authenticate: Bearer realm="oemcp",
resource_metadata="https://mcp-server.example.com/.well-known/oauth-protected-resource"
```

## Client discovery flow

The following steps describe how a client discovers and uses resource metadata:

| Step | Description |
|---|---|
| Initial request | Client attempts to access the resource without a token. |
| 401 response | Server returns a `WWW-Authenticate` header with the metadata URL. |
| Metadata discovery | Client fetches the `/.well-known/oauth-protected-resource` endpoint. |
| Authorization server discovery | Client uses the `authorization_servers` field from metadata. |
| Token acquisition | Client obtains an OAuth token from the authorization server. |
| Retry with token | Client retries the original request with a Bearer token. |

### Configuration parameters

The following table lists key configuration parameters for protected resource metadata:

| Parameter | Description |
|---|---|
| `bearer_methods_supported` | Specifies supported token presentation methods such as header, body, or query. |
| `require_metadata_on_401` | Determines whether the metadata URL is included in `WWW-Authenticate` headers. |
| `resource_documentation` | URL to human-readable API documentation. |
| `resource_policy_uri` | URL to the token processing policy document. |
| `revocation_endpoint` | OAuth token revocation endpoint. |
| `introspection_endpoint` | OAuth token introspection endpoint. |

## Token exchange for downstream services

Token exchange is used when an upstream user token cannot be forwarded directly to a downstream service. This situation occurs when the downstream service requires a different audience, a shorter time-to-live (TTL), or sanitized scopes. Token exchange ensures secure and appropriate token handling between services.

The following table summarizes the key guidelines for implementing token exchange:

| Guideline | Description |
|---|---|
| Always use HTTPS | All token exchange operations must occur over HTTPS to protect token confidentiality and integrity. |
| Fail closed | If token exchange fails, the system must reject the request rather than fallback to using the original unexchanged token. This prevents unauthorized access. |
| Scope separation | The exchanged token should include only the scopes required by the downstream service. This minimizes privilege exposure and enforces least privilege principles. |

For more information about token exchange for PAS for OpenEdge integration, see OAuth mode on page 167.

## TLS and mTLS for secure communication

Transport Layer Security (TLS) and mutual TLS (mTLS) are critical for securing communication between services. TLS ensures encrypted transport, while mTLS adds client certificate enforcement for stronger mutual trust. This topic explains how TLS and mTLS apply to the MCP server and downstream services.

### Server transport security

The MCP interface uses TLS to secure all communication between clients and the server. TLS provides encryption and integrity for data in transit.

- **TLS**—Wraps the MCP interface to protect against eavesdropping and tampering.
- **mTLS**—Adds client certificate enforcement, ensuring that both the server and client authenticate each other. This creates a stronger trust model.

### Downstream TLS and mTLS configuration

Downstream services can be configured independently for TLS or mTLS. This flexibility allows you to supply a separate client identity when calling upstream APIs.

- **Independent configuration**—Downstream TLS and mTLS settings do not depend on MCP server settings.
- **Separate client identity**—You can provide a unique client certificate for upstream API calls to differentiate identities across services.

## Response guard strategy

Response guards control how the system handles large or unexpected payloads to maintain predictable failure behavior.

The default response guard `block` blocks oversized payloads to maintain predictable failure behavior. You should only enable `trim` when client applications can safely handle truncated arrays without causing functional issues. Never disable the response guard in a production environment unless upstream payload sizes are strictly controlled and audited to ensure compliance and prevent unpredictable failures.

## Logging hygiene

Logging hygiene ensures sensitive information is protected while maintaining useful diagnostic data. Here are important recommendations for ensuring proper logging practices:

- Sensitive headers, such as service account (SA) tokens, must always be masked in logs to prevent exposure of credentials.
- User tokens can be optionally masked based on the configured log level policy to balance security with troubleshooting needs.
- Avoid enabling payload logging by setting `include_payloads` to true except during short, controlled debug sessions, as prolonged payload logging can expose sensitive data and increase compliance risks.

## Key and token rotation cadence

Regular rotation of keys and tokens is essential for maintaining security, reducing the risk of compromise, and ensuring cryptographic agility.

The following table summarizes the recommended rotation intervals for service account tokens, key pairs, and TLS certificates:

| Item | Recommended cadence | Rationale |
|---|---|---|
| Service account JWT | 30–90 days | Limits the replay window and reduces exposure if a token is compromised. |
| Service account key pair | 90–180 days | Maintains key hygiene and supports cryptographic agility. |
| TLS certificate | Based on CA policy (typically 90 days) | Aligns with industry norms and encourages automation of certificate renewal. |

# 12

# Manage prompts and tooling

Prompts provide curated instruction and context chunks for downstream LLM orchestration layers. The server enforces a size guard and provides helper tools for prompt selection and introspection. A prompt is scope-filtered to ensure visibility only to authorized users and can coordinate multiple tools for complex workflows. Prompts are loaded from the `/work/server/prompts` directory configured by `assistant.prompts.dir`. For configuration details, see Configure the prompt directory and size guard on page 213.

For example, the following prompt acts as a server-provided workflow guide for generating structured, AI-driven product recommendations for customers and operations in the `Sports2020` database:

```
# Product Recommendations
Generate concise, actionable product recommendations for Sports2020
customers and operations.


## Instructions
Analyze the requested product/category and provide BRIEF
recommendations in this format:


### [Product Name]

**Priority**: [High/Medium/Low]
**Stock Status**: [OnHand / Allocated / Available]
**Action**: [1-2 sentence recommendation]
**Cross-Sell**: [2-3 complementary items]
**Supplier**: [Name, Discount %]


## Key Query Patterns
**For customer recommendations**:
1. `getBECustomer?filter=Name MATCHES "{name}"` → Get CustNum
2. `getBEOrder?filter=CustNum={id}` → Analyze order history
3. `getBEItem?filter=Category1="{category}" AND
OnHand > 10` → Find available items


**For reorder needs**:
`getBEItem?filter=OnHand < ReOrder AND
OnOrder = 0` → Items needing immediate reorder


**For popular items**:
Aggregate `getBEOrderLine` frequency + `getBEInvoice` revenue
to identify top performers


## Output Guidelines
- Keep response under 200 words
- Focus on top 2-3 actionable insights
- Include specific item numbers and prices
- Prioritize immediate actions over detailed analysis
```

For details, see the following topics:

- Configure the prompt directory and size guard
- Use helper tools for prompt management
- Secure prompts with scopes
- Add new prompts
- Selection strategies for prompts
- Failure modes and resolutions

# Configure the prompt directory and size guard

Prompts are configured through the `assistant.prompts.dir` property. The default directory is `./prompts`. A cumulative character budget is applied to protect the memory footprint. If the budget is exceeded, the system fails during startup. This fail-fast approach ensures stability.

## Configuration details

The following table summarizes the configuration properties:

| Property | Default value | Description |
|---|---|---|
| `assistant.prompts.dir` | `./prompts` | Specifies the directory where prompts are stored |
| Size guard | Applied | Enforces a cumulative character budget |

## Notes

- Keep prompts small and modular.
- Ensure the cumulative size does not exceed the configured budget.
- Validate prompt size before startup to prevent failure.

# Use helper tools for prompt management

The system provides helper tools to inspect and manage prompts. These tools allow you to view all available prompts and their token estimates.

Use the `select_prompts` helper tool to returns all prompts along with token estimates. Both mid and conservative estimates are included. This tool does not apply a selection strategy.

**Note:** Exact tool IDs depend on the implementation. To verify tool IDs, run the following command:

```
./mcpgen client <profile_name>
```

## Validation tips for prompt size

Use the output of `select_prompts` to verify that no single prompt is unexpectedly large. Keep each individual prompt between 4 KB and 8 KB unless there is a clear justification for exceeding this range.

### Example output

The following example shows the output of the `select_prompts` helper tool:

```
{
  "query": "status guidance",
  "context_window": 8192,
  "selected_prompts": [
    {"name": "auto_assistant", "tokens_mid": 120, "tokens_conservative": 150},
    {"name": "product_recommendation", "tokens_mid": 90, "tokens_conservative": 110}
  ],
  "total_selected_tokens_mid": 210,
  "notes": "All registered prompts returned with token estimates."
}
```

## Secure prompts with scopes

Scopes are used to secure prompts by requiring explicit authorization. Without the required scope, the tools remain hidden.

The following example shows how to configure scopes for prompt tools:

```
{
  "security": {
    "authorization": {
      "tools": {
        "default": ["mcp_access"],
        "select_prompts": ["prompts.read"]
      }
    }
  }
}
```

**Note:** If the scope is not defined, the tools are not visible to users.

## Add new prompts

The following steps show how to add new prompts:

1. Place a Markdown file in the prompts directory. Use a short descriptive name and the `.md` extension.
2. Do not embed secrets in the prompt file. Treat prompts as assets that can eventually be made public.
3. Restart the server to load the new file. Hot reload is not supported.

## Selection strategies for prompts

The following selection strategies are planned but not yet active:

- **Size-first strategy**—Sort prompts by token or character size and pick the top N prompts that fit within the configured budget.
- **Tag-filtered strategy**—Use filename prefixes or embed lightweight front matter for categorization. This feature is planned for future releases.
- **Hybrid strategy**—Apply a tag filter first, then enforce the size budget, and finally apply explicit priority ordering.

### Current behavior

The current implementation returns every prompt with token estimates. Downstream agents can apply their own budgeting heuristics later.

## Failure modes and resolutions

The following table lists common failure modes, their causes, and recommended actions:

| Issue | Cause | Action |
|---|---|---|
| Startup failure due to prompt budget | Aggregate size exceeds the configured guard | Remove or shrink prompts. Split large documents. |
| Tool hidden | Missing scope | Adjust token scopes or update configuration mapping. |
| Empty prompt plan | `--no-prompt` flag used or directory empty | Add prompts or omit the `--no-prompt` flag. |
| Selection returns empty set | Over-restrictive filters | Loosen filter criteria or inspect filenames. |

For details on how prompt scopes and context enrichment interact during discovery, see **OpenAPI visibility and context management**.

# 13

# OpenAPI visibility and context management

This section describes optional visibility enrichment features and the per-tool context management system. These features improve tool discoverability and operator guidance without changing business logic or security posture. All features are opt-in and size-bounded to prevent excessive description growth.

For details, see the following topics:


- Define goals and guiding principles
- Visibility and context features
- Generate filter examples
- Provide parameter examples
- Manage per-tool context
- Configuration examples for visibility and context features
- Test visibility and context enrichment features
- Quick checklist for enabling context features

# Define goals and guiding principles

Visibility enrichment is designed to:

- Provide small, deterministic *filter* examples so that a user interface (UI) can quickly suggest how to use complex filtering.
- Supply non-intrusive parameter examples for existing query or path parameters. These examples never create new parameters, never overwrite explicit examples, and skip sensitive names.
- Inject lightweight per-tool context snippets from a local directory. Ordering is controlled by priority and capped through configuration.

## Design principles

The following design principles define how visibility enrichment and context management features operate to ensure consistency and predictability:

- **Read-only**—Enrich specification descriptions only. Do not modify schemas except to add examples.
- **Deterministic**—The same specification and context directory always produce identical output.
- **Safe defaults**—Features are disabled unless explicitly enabled. Parameter examples are added only when requested.
- **Size-bounded**—Apply character and line truncation to prevent oversized context from leaking into prompt windows.

## Review key properties

The following table summarizes important configuration properties for OpenAPI visibility and context management and explains their purpose:

| Property | Purpose | Default or guidance |
|---|---|---|
| `add_filter_examples` | Add consolidated `Filter Examples:` (or `Filter:`) and sets an example on an existing filter parameter. | Disabled unless enabled. Uses first server URL and first 8 fields (maximum 3 predicates). |
| `add_param_examples` | Populates examples for existing query or path parameters. Skips sensitive names. | Disabled unless enabled. Non-destructive. |
| `per_tool_context_enabled` | Enables context injection from `<operationId>.md` or `.txt` files. | Disabled by default. Opt-in feature. |
| `per_tool_context_char_limit` | Sets maximum characters per per-tool snippet. | If not set, the internal default is 10000 characters, but it is capped at 2000 characters. For clarity and predictable behavior, set the value explicitly to 2000 or a lower number. |
| `per_tool_context_max_files` | Limits number of operations receiving context after priority sort. | Unlimited if unset or less than or equal to zero. |
| `global_context_char_limit` | Sets maximum characters pulled from `global.md` before distribution to all operations. | Default 10000. Recommended range is 3000–6000 for clarity. |
| `context_dir` | The directory is used to read `global.md`, per-tool files, and `_index.json`. | The system automatically detects the directory in the following order:<br>**1.** An explicitly specified path<br>**2.** `./context`<br>**3.** `repo/context/` |
| Catalog tools | Provides runtime introspection utilities, such as `list_all_tags`, `get_tool_schema`, and `get_filter_fields`. | Scope-gated if exposing internal schemas. |

**Note:** Only `priority` is honored per operation entry in `_index.json.operations`. Formatting keys in this file are ignored.

# Visibility and context features

A feature refers to an optional capability or enhancement that can be enabled or disabled in the OpenAPI visibility and context management system. Each feature provides a specific function, such as adding filter examples, injecting per-tool context, or populating parameter examples. These features do not change the core business logic or security posture; they only enrich the API specification for better usability and discoverability.

The following table lists available features, configuration keys, and their effects:

| Feature | Configuration key | Effect | Note |
|---|---|---|---|
| Filter examples | `openapi_visibility.add_filter_examples` | Adds a consolidated `Filter Examples:` line or `Filter:` for single field, and sets example on existing `filter` query parameter. | Activates only if a `filter` query parameter exists. It supports up to three predicates derived from the first eight output fields. It does not include per-example usage lines. |
| Parameter examples | `openapi_visibility.add_param_examples` | Sets example values on existing query or path parameters that do not already have examples. | Sensitive names such as `token`, `secret`, and `password` are skipped. This feature does not create new parameters. |
| Per-tool context | `openapi_visibility.per_tool_context_enabled` | Appends a `Context:` block or line for operations that have a matching `.md` or `.txt` file. | This feature is disabled by default and must be explicitly enabled. |
| Max context files | `openapi_visibility.per_tool_context_max_files` | Limits the number of operations that receive context after applying priority sorting. | If this property is omitted or set to a value less than or equal to zero, the number of files is unlimited. |
| Context character limit | `openapi_visibility.per_tool_context_char_limit` | Sets the global maximum number of characters allowed per per-tool context snippet. | The internal hard ceiling is 2000 characters. If the property is unset, the default is 10000 characters, but it is capped at 2000. Set this property explicitly to 2000 or lower for clarity. |
| Global context limit | `openapi_visibility.global_context_char_limit` | Sets the maximum number of characters appended from `global.md` to the `info.description` and each operation. | The default value is 10000 characters. It is recommended to lower this value to between 3000 and 6000 characters for clarity. |
| Context directory | `openapi_visibility.context_dir` | This feature specifies the directory that is scanned for context assets. | The default search order is:<br>**1.** An explicitly specified directory<br>**2.** `./context`<br>**3.** `repo/context/` |

| Feature | Configuration key | Effect | Note |
|---|---|---|---|
| Catalog and schema introspection | Tools such as `list_all_tags`, `get_tool_schema`, and `get_filter_fields` | These tools surface tags and resolved schemas for runtime introspection. | Assign read scopes if the information is sensitive. |

# Generate filter examples

When `add_filter_examples` is enabled, the system generates consolidated filter examples based on schema information. The following example shows an operation description that includes a JSON line:

```
Schema: {"output": {"fields": [{"name": "id", "type": "integer"}, {"name": "name",
"type": "string"}]}}
```

Enabling `add_filter_examples` results in this consolidated output:

```
Filter Examples:
https://api.example.com/items?filter=id>0
https://api.example.com/items?filter=id>0 AND name="value"
```

If only one field is present:

```
Filter: id>0
```

Follow these rules for filter example generation:

- A filter query parameter must already exist in the operation or at the path level for this feature to work.
- Multi-field mode supports up to three predicates combined into one line.
- Each predicate is rendered as a full URL using the first server entry as the base.
- No separate usage lines are added. The header `Filter Examples:` precedes the URLs.
- The first predicate becomes the example value on the filter parameter.
- Idempotent behavior ensures re-running enrichment does not duplicate examples or append additional URL lines.

# Provide parameter examples

When `parameter_examples` is set to true, the enrichment process inspects existing path and query parameters, including parameters defined at both the path level and the operation level, resolving any local `$ref` references. For each parameter that does not already have an example, the system applies the following rules:

- **Derive example values using heuristics**
  - For `integer` parameters, the example value is `1`.
  - For `string` parameters with a `format:date`, the example value is `2024-01-01`.
  - Other types follow similar simple heuristics.
- **Skip sensitive parameter names**

  Parameters that contain authentication-related substrings such as `auth`, `token`, `secret`, or `password` are excluded from example generation.
- **Do not overwrite existing examples**

  The system does not replace any `example` fields that already exist at the parameter level or schema level.

## Purpose of parameter examples

Manage per-tool context for API operationsManage per-tool context for API operationsThe purpose of parameter examples is to provide helpful hints in the user interface. These examples are intended to improve usability and clarity without imposing any business-specific semantics.

# Manage per-tool context

Per-tool context enriches operation descriptions with curated snippets from a context directory. These snippets provide domain-specific hints such as field semantics, guardrails, and usage caveats. The goal is to improve guidance for LLMs and UI consumers without altering schemas or business logic.

## Per-tool context enrichment

When enabled, per-tool context appends a `Context` block to the operation description. The block can preserve newlines or appear as a single line.

For example,

- In newline-preserving mode, the `context` block appears as:

```
Context:
Allowed status values: NEW | RUNNING | COMPLETE
Max page size: 100
… (truncated)
```

- In single-line mode, the `context` block appears as:

```
Context: Allowed status values: NEW | RUNNING | COMPLETE … (truncated)
```

## Activate per-tool context

The following steps describe how to activate per-tool context:

1. Set the feature gate `openapi_visibility.per_tool_context_enabled` to `true`. By default it is set to `disabled`.
2. Resolve the context directory in the following order:
   a. The context directory uses the explicit value of the `openapi_visibility.context_dir` property if it is provided.
   b. If the property is not provided, it checks for a `./context` directory relative to the current working directory.
   c. If neither of the above exists, it falls back to the `context/` directory located in the repository source tree.
3. For each operation with an `operationId`, look for `<operationId>.md` and then `<operationId>.txt` in the context directory.
4. Sort candidate files by descending priority and then by `operationId`. Truncate the list to `per_tool_context_max_files` if set.
5. Append each selected file according to global defaults if not already present. Global context supports only `global.md`. Other filenames are ignored.

### Append global context from `global.md`

If `global.md` exists in the resolved context directory, its content is appended to:

- `spec.info.description` for a high-level overview with the help of the annotator.
- Every operation description as a `Global Context:` block or line when context features are enabled.

The following table describes configuration tuning settings that control global context size and formatting behavior:

| Setting | Description |
|---|---|
| Size limit | The `openapi_visibility.global_context_char_limit` property sets the size limit for global context. The default value is 1500 characters, and the maximum allowed is 2000 characters. |
| Formatting | Formatting respects the settings defined in `_index.json.defaults` for newline preservation and truncation. |

You can follow this guidance on recommended content that should be included in the `global.md` file when using the Global Context feature in OpenAPI visibility and context management:

- Data freshness—State how often the inventory or data is updated.
- Pagination—Provide limits and usage hints for pagination.
- Rate limiting—Indicate throttling behavior for large queries.

The following example shows the recommended content in concise bullet points:

```
Data freshness: Inventory updated every 15 minutes.
Pagination: Top max 500; use Skip for offsets.
Rate limiting: Burst queries >10k items may throttle.
```

**Note:** Lengthy procedural guides, secrets, or large tables.

### Control formatting with `_index.json`

The `_index.json` file inside the context directory provides optional global formatting controls and operation priorities. For example:

```
{
  "defaults": {
    "priority": 0,
    "preserve_newlines": false,
    "max_lines": 8,
    "char_limit": 400
  },
  "operations": {
    "GetItems": { "priority": 80 },
    "SearchItems": { "priority": 60 }
  }
}
```

**Note:**

- Only `priority` is honored per operation. Legacy keys such as `preserve_newlines`, `max_lines`, and `char_limit` are ignored.
- Defaults apply globally to all operations.

### Apply formatting and truncation rules

You can adjust how context snippets are formatted and truncated by changing the global defaults in `_index.json.defaults`. The following table explains each setting:

| Setting | Data type | Description |
|---|---|---|
| `preserve_newlines` | Boolean | If `false`, the context is collapsed into a single line. If `true`, original lines are preserved up to the value of `max_lines`. |
| `max_lines` | Integer | Specifies the maximum number of lines retained when `preserve_newlines` is `true`. If the limit exceeds, the system adds … `(truncated)` to indicate truncation. |
| `char_limit` | Integer | Defines the maximum number of characters before line splitting occurs. The hard upper bound is 2000 characters. Truncation respects newline boundaries whenever possible to avoid mid-line cuts and appeands … `(truncated)`. |

### Apply priority and limits

Operations are sorted by priority. If `per_tool_context_max_files` is set, only the first N operations are enriched. This allows staged rollouts, such as highlighting the top 10 most-used operations.

### Ensure idempotency and avoid collisions

When enriching API specifications with per-tool context, the system must ensure that the process does not create duplicate entries or introduce inconsistencies.

Idempotency ensures that running the enrichment multiple times produces the same result without duplication. If the enrichment process is executed more than once, the system checks for an identical formatted block in the operation description. If the block already exists, it is skipped. This ensures that repeated runs do not add duplicate context.

Collision avoidance prevents near-duplicate content from being added when similar information already exists in the operation description. Before adding new context, the system compares the first line of the raw file (up to 40 characters) with the existing description. If this snippet already appears, enrichment is skipped. This prevents adding content that is too similar to manually documented details.

## Review logging behavior during enrichment

When the enrichment process runs, the system generates log messages to indicate actions taken:

- If per-tool context is successfully added, the log message is:

  ```
  Visibility: added per-tool context to X operations (dir=...)
  ```

  This message confirms the number of operations enriched and the directory used.

- If unsupported per-operation keys are removed, the log message is:

  ```
  Visibility: stripped N unsupported per-operation context keys
  ```

  This message indicates how many invalid keys were detected and stripped during processing.

## Avoid common pitfalls when using per-tool context

The following table lists common pitfalls, what happens when they occur, and how to resolve them:

| Pitfall | Symptom | Fix |
|---|---|---|
| Forgetting to enable `per_tool_context_enabled` | No context lines appear in the operation descriptions | Set the `per_tool_context_enabled` flag to **true** in the configuration. |
| Using per-operation formatting keys | Formatting keys are silently ignored | Move all formatting settings to the `defaults` section in `_index.json`. |
| Large context files | Context is aggressively truncated | Refactor the content into concise bullet points to fit within limits. |
| Duplicate run of enrichment | No duplication occurs (expected behavior) | This is working as intended. No action is required. |
| Missing `operationId` in operations | No context is injected | Ensure that every operation in the specification includes an `operationId`. |

# Configuration examples for visibility and context features

This topic provides sample configuration snippets that demonstrate how to enable OpenAPI visibility features and control per-tool context behavior.

## Enable all visibility features

The following example shows how to enable all visibility enrichment features, including filter examples, parameter examples, and per-tool context. It also sets limits for context files and character size. Use this configuration when you want to activate all visibility features::

```
{
  "openapi_visibility": {
  "add_filter_examples": true,
  "add_param_examples": true,
    "per_tool_context_enabled": true,
    "per_tool_context_max_files": 12,
    "per_tool_context_char_limit": 600,
    "context_dir": "./context"
  }
}
```

## Enable only per-tool context

The following example shows a minimal configuration that enables per-tool context enrichment without activating other visibility features. Use this configuration when you want to enable only per-tool context:

```
{
  "openapi_visibility": {
    "per_tool_context_enabled": true,
    "context_dir": "./context"
  }
}
```

## Example _index.json file with operation priorities

The `_index.json` file allows you to control global formatting defaults and set priorities for specific operations. Higher priority values ensure that those operations are enriched first. The following example shows how to set global defaults and elevate two operations using priority values:

```
{
  "defaults": {
    "priority": 0,
    "preserve_newlines": true,
    "max_lines": 5,
    "char_limit": 500
  },
  "operations": {
    "CreateOrder": { "priority": 90 },
    "SearchOrders": { "priority": 70 }
  }
}
```

# Test visibility and context enrichment features

This topic provides guidance on unit and integration tests that validate enrichment logic, formatting, and configuration behavior.

## Unit testing for enrichment logic

The following unit tests verify generation logic and formatting edge cases:

- `tests/unit/test_openapi_enrich.py`
- `tests/unit/test_openapi_per_tool_context.py`

These tests ensure that filter examples, parameter examples, and per-tool context are generated correctly and handle edge cases such as truncation and newline preservation.

## Integration testing for context management

The `tests/integration/test_per_tool_context_integration.py` integration test validates end-to-end behavior for per-tool context enrichment. This test covers:

- Priority sorting
- Truncation behavior
- Maximum file limits
- Disabled feature flags
- Ignoring unsupported per-operation formatting keys

# Quick checklist for enabling context features

The following table summarizes common goals and the corresponding actions:

| Goal | Action |
|---|---|
| Add contextual hints | Create `<operationId>.md` in the context directory. |
| Control order | Assign priority in `_index.json`. |
| Reduce noise | Set `per_tool_context_max_files` to a small number. |
| Prevent line bloat | Tune `char_limit` and `max_lines`. |
| Generate multi-field filter examples | Ensure the output field list exists in the `Schema:` line and a filter parameter is present. |

# 14

# Export and harden production deployment

This topic explains how to transform a development profile into a secure and deployable production artifact. It provides clear objectives and mechanisms to ensure production readiness.

This following table describes the key objectives for creating hardened production artifacts and the mechanisms to achieve them:

| Objective | Mechanism |
|---|---|
| Clean production artifacts | Use `--bundle prod` during export |
| Reduce attack surface | Remove client utilities, verbose configurations, and private keys |
| Ensure configuration clarity | Provide a minimal `mcp_server_config.json` with only overrides |
| Maintain secret hygiene | Externalize tokens and keys using Vault or key management system (KMS) |

For details, see the following topics:

- Export modes
- Hardening actions
- Export process
- Quality, security, and compliance checklist for MCP deployment
- Run the application in production
- Scaling considerations

- Observability in production
- Security and performance checklist
- Rotate security artifacts in production
- Disaster recovery guidelines for MCP servers
- Common failure modes and resolutions

# Export modes

Export modes define how a development profile is converted into a deployable artifact. They determine the level of portability, security, and readiness for production environments. Choosing the correct export mode ensures that the artifact meets the requirements of its target environment.

The export process creates a deployable artifact from a development profile. There are two primary modes:

| Command | Purpose |
|---|---|
| `./mcpgen export <profile_name>` | Creates an artifact for development portability |
| `./mcpgen export <profile_name> --bundle prod` | Creates a hardened artifact for production deployment (recommended) |

For more information about the optional flags that can be used during export, see Export profiles for production on page 127. It is recommended to use these flags applied sparingly to avoid introducing security risks.

# Hardening actions

You apply hardening actions to a profile before export or during its transition from development to production. Hardening ensures that the deployment artifact is secure, optimized, and aligned with production standards.

The `./mcpgen harden <profile_name>` command can be executed either before export or on a development profile that is evolving toward production readiness. Hardening removes unnecessary components, enforces security measures, and improves supply chain integrity.

The following actions are applied during hardening:

| Action | Description |
|---|---|
| Drop verbose configuration | Prevents leaking default settings that may reveal unused features |
| Remove client directory | Avoids accidental misuse of embedded Service Account tokens |
| Replace image tag with digest | Ensures supply chain integrity by using immutable image references |

## Production configuration examples

The following examples demonstrate different levels of hardening for production environments.

- This example shows full production configuration with authentication, TLS, and all hardening features:

```
{
  "api": {
    "openapi": {
      "spec_file": "/work/server/openapi_oeping.yml"
    }
  },
  "security": {
    "authentication": {
      "enabled": true,
      "service_account": {
        "enabled": true,
        "required": true,
        "mode": "jwt",
        "header": "X-OEMCP-SERVICEACCOUNT",
        "issuer": "https://your-auth-server.com",
        "required_scopes": ["mcp_access"],
        "jwks_uri": "https://your-auth-server.com/.well-known/jwks.json",
        "algorithms": ["RS256"],
        "jwks_cache_seconds": 300
      },
      "user_auth": {
        "enabled": false
      }
    }
  },
  "runtime": {
    "max_request_bytes": 10240,
    "server": {
      "name": "openedge-mcp-production",
      "transport": {
        "type": "streamable-http",
        "host": "0.0.0.0",
        "port": 8500,
        "path": "/http",
        "tls": {
          "certfile": "/work/certs/localhost.pem",
          "keyfile": "/work/certs/localhost-key.pem",
          "require_client_cert": false
        }
      }
    },
    "http": {
      "base_url": "http://3.13.139.192:8810",
      "response_guard": {
        "enabled": true,
        "mode": "trim",
        "max_array_items": 50,
        "max_response_bytes": 1048576
      },
      "limits": {
        "max_connections": 100,
        "max_keepalive_connections": 20
      },
      "tls": {
        "certs_dir": "/work/certs",
        "insecureSkipVerify": false,
        "ignoreHostVerification": false
      }
    },
    "middleware": {
      "error_handling": {
        "enabled": true,
        "transform_errors": true,
        "include_traceback": false
      },
      "logging": {
        "enabled": true,
        "structured": true,
        "include_payloads": false
```

```
      },
      "request_timeout": {
        "enabled": true,
        "timeout_ms": 500
      },
      "method_validation": {
        "enabled": true
      },
      "circuit_breaker": {
        "enabled": true,
        "error_threshold": 0.5,
        "window_seconds": 60,
        "min_requests": 10
      },
      "rate_limiting": {
        "enabled": true,
        "max_requests_per_second": 10.0,
        "burst_capacity": 20,
        "global": false
      },
      "audit": {
        "enabled": true,
        "file": "/var/log/mcp/audit.log",
        "max_bytes": 10485760,
        "backup_count": 5,
        "mode": "append"
      }
    },
    "tools": {
      "param_validation": {
        "enabled": true,
        "max_total_bytes": 32768,
        "max_string_length": 4096,
        "patterns": [
          "javascript:",
          "data:text/html",
          "<script",
          "on[a-z]+=",
          "file:",
          "\\.\\./",
          "\\.\\.\\\\"
        ]
      }
    }
  },
  "assistant": {
    "prompts": {
      "dir": "/work/server/prompts"
    }
  },
  "observability": {
    "logging": {
      "level": "INFO",
      "json": true,
      "loggers": {
        "middleware.circuit_breaker": "WARNING",
        "middleware.timeout": "WARNING",
        "http.outbound": "INFO",
        "auth": "INFO"
      }
    }
  }
}
```

- This example shows minimal hardening configuration:

```
{
  "api": {
    "openapi": {
      "spec_file": "openapi.yml",
      "base_url": "https://api.internal.example"
    }
  },
  "security": {
    "authentication": {
      "enabled": true,
      "service_account": {
        "enabled": true,
        "required": true,
        "mode": "jwt",
        "header": "X-OEMCP-SERVICEACCOUNT",
        "issuer": "https://auth.example.com",
        "required_scopes": ["api.read", "api.write"],
        "jwks_uri": "https://auth.example.com/.well-known/jwks.json",
        "algorithms": ["RS256", "ES256"],
        "jwks_cache_seconds": 300
      },
      "user_auth": {
        "enabled": false
      }
    }
  },
  "runtime": {
    "max_request_bytes": 10240,
    "http": {
      "response_guard": {
        "enabled": true,
        "mode": "block",
        "max_array_items": 50,
        "max_response_bytes": 524288
      },
      "limits": {
        "max_connections": 100,
        "max_keepalive_connections": 20
      },
      "tls": {
        "certs_dir": "/etc/mcp/downstream-certs",
        "insecureSkipVerify": false,
        "ignoreHostVerification": false
      },
      "accept_encoding": "none"
    },
    "middleware": {
      "error_handling": {
        "enabled": true,
        "transform_errors": true,
        "include_traceback": false
      },
      "logging": {
        "enabled": true,
        "structured": true,
        "include_payloads": false
      },
      "request_timeout": {
        "enabled": true,
        "timeout_ms": 500
      },
      "method_validation": {
        "enabled": true
      },
      "circuit_breaker": {
        "enabled": true,
        "error_threshold": 0.5,
        "window_seconds": 60,
        "min_requests": 10
```

```
      },
      "rate_limiting": {
        "enabled": true,
        "max_requests_per_second": 5.0,
        "burst_capacity": 10,
        "global": false
      },
      "audit": {
        "enabled": true,
        "file": "/var/log/mcp/audit.log",
        "max_bytes": 1048576,
        "backup_count": 3,
        "mode": "append"
      }
    },
    "tools": {
      "param_validation": {
        "enabled": true,
        "max_total_bytes": 32768,
        "max_string_length": 4096,
        "patterns": [
          "javascript:",
          "data:text/html",
          "<script",
          "on[a-z]+=",
          "../"
        ]
      }
    }
  },
  "observability": {
    "logging": {
      "level": "INFO",
      "json": true
    }
  }
}
```

## Key hardening features to enable

The following features should be enabled or configured for production environments:

| Feature | Default behavior | Recommendation |
|---|---|---|
| Request timeout | 500 ms, enabled by default | Keep default or adjust based on workload |
| Method validation | Enabled by default | Required for all deployments |
| Circuit breaker | Opt-in | Recommended for production environments |
| Rate limiting | Opt-in | Recommended for public-facing deployments |
| Payload size limits | 10 KB default | Adjust if handling larger payloads |
| Connection pool limits | Not specified | Configure based on expected concurrency |
| Response guards | Not specified | Enable to prevent unsafe responses |

# Export process

This topic explains how to create production-ready bundles using the export process. It also highlights the differences between development and production exports and the security features included in production bundles.

## Create a production bundle

Use the `export` command to generate a complete deployment package for production environments. This package contains all necessary components for secure deployment. Run the following command to create a deployment package in the `exports/<profile>/` directory:

```
./mcpgen export <profile_name> --bundle prod
```

For more information about the `export` command, see Export profiles for production on page 127. To know more about why and how to export artifacts for production environments, see Prepare a production-ready export on page 60.

## Recommended production layout

A well-structured production layout ensures clarity, security, and ease of deployment. The layout minimizes configuration surface, excludes sensitive data, and provides only the components required for production environments.

The following code block shows the recommended directory structure for a production export bundle:

```
exports/<profile_name>/
  docker-compose.mcp.yml
  server/
    mcp_server_config.json  # minimal; no inline keys
  certs/ #public certs only
  profile.json #sanitized profile metadata
  run.sh #optional helper
```

Secrets must be mounted securely during runtime instead of being included in the export bundle.

- Example mount path: `/run/secrets/sa_public.key`
- Set the environment variable `SERVICE_ACCOUNT_PUBLIC_KEY_FILE` to reference the mounted path.

# Quality, security, and compliance checklist for MCP deployment

Before promoting an MCP deployment to production, it is critical to validate that all components, such as OpenAPI specifications, authentication, authorization, routing, and hardening controls, meet security and compliance standards. This checklist serves as a quality gate to ensure there are no gaps across individual configurations and that the deployment is robust, secure, and maintainable.

## OpenAPI specification quality

The OpenAPI specification must be clean and expose only intended operations. Ensure the OpenAPI specification meets these conditions:

| Requirement | Description |
|---|---|
| Unique `operation ID` per opertation | Each operation must have a unique and stable `operationId`. |
| Complete descriptions | Include descriptions, component schemas, pagination details, and error schemas. |
| Lint clean | Validate that the specification passes lint checks. |
| Expose only intended operations | Confirm that only the required endpoints are exposed. |

## Authentication and scopes

Use the following checklist to verify authentication and scope requirements:

| Requirement | Description |
|---|---|
| Service account configuration | Set up `service_account.*` for production. Optionally, you can configure user token exchange (`user_auth.*`). |
| Scope definition | Define scopes in `security.authorization.tools`. |

## Downstream hardening

Apply security and performance controls:

| Requirement | Description |
|---|---|
| Limit surfaced routes | Use `INCLUDE` maps. Default to read-only routes. |
| Audit and scope writes | Ensure write operations are audited and scoped. |
| Paginate large responses | Bound large responses to prevent overload. |

## TLS and mTLS configuration

Secure transport for inbound and outbound traffic:

| Requirement | Description |
|---|---|
| Inbound TLS | Configure `runtime.server.transport.tls.*` and enable `require_client_cert` for mTLS if needed. |
| Outbound TLS | Trust CA using `runtime.http.tls.ca_bundle`; provide client certificate if required. |

## JWKS and key rotation

Implement secure key management:

| Requirement | Description |
|---|---|
| JWKS URI | Use `jwks_uri` with short cache duration. |
| Algorithm restrictions | Restrict algorithms if desired. |

## Integrated gate purpose

This gate verifies that all security, routing, and hardening controls work together before enabling user access. It ensures there are no gaps between individual checklists, and verifies:

- **Authentication**—Valid tokens are accepted, and invalid or expired tokens are rejected with HTTP status codes 401 or 403.
- **Authorization and scopes**—Tools without granted scopes disappear from discovery, and scoped write tools return HTTP 403 when the required scope is missing.
- **Transport security**—All endpoints are reachable only over TLS, and the mTLS handshake succeeds or fails as configured.
- **Response guardrails**—Size and item count limits trigger graceful truncation or policy errors instead of causing server failures.
- **Route mapping**—Only intended PAS for OpenEdge endpoints are exposed, and any removed paths return HTTP 404.
- **Export integrity**—The deployment is built using the `--bundle prod` option, and container references confirm clean production artifacts.
- **Logging and privacy**—Logs do not contain sensitive fields or raw tokens, and debug logging is disabled in production.
- **Observability hooks**—Authentication failures, guard rejections, and rate limits generate structured log entries or metrics for alerting.

## Actionable gate checklist

Use this checklist to validate that all integrated security, routing, and hardening controls function correctly before promoting the deployment to production:

- Negative authentication tests return only 401/403 (never 200 or 500).
- Scope removal hides tools from discovery and returns 403 if invoked manually.
- Oversized response triggers response guard with sanitized error and log entry.
- Disallowed routes return 404.
- All endpoints are accessible only through HTTPS. HTTP requests fail.
- mTLS blocks clients without valid certificates and allows those with valid certs.
- Logs are free of secrets and tokens; request IDs traceable end-to-end.

**Gate outcome**: Promote only when every check passes. Failures must be remediated in the relevant section, such as authentication, TLS, hardening, JWKS.

### Rotation matrix

Use this rotation matrix to plan and enforce regular key, certificate, and configuration updates across all critical areas to maintain security and compliance:

| Area | Keys | Artificate/scope | Rotation cadence/trigger |
|---|---|---|---|
| OpenAPI specification | `api.openapi.*` | OpenAPI YAML | Whenever there is a schema change that requires a version bump |
| Authentication and authorization | `security.authentication.*`, `security.authorization.*` | Service account keys, JWKS, token exchange endpoint | Service account keys every 30–90 days, and ensure user tokens remain short-lived |
| Routes and HTTP configuration | `api.routes.*`, `runtime.http.*` | PAS for OpenEdge upstream endpoints | Whenever there is a new release or when upstream endpoints change |
| Transport Layer Security (TLS) | `runtime.server.transport.tls.*`, `runtime.http.tls.*` | TLS certificates and CA bundle | Every 60–90 days or before certificate expiry |
| JSON Web Key Set (JWKS) | `jwks_uri` | Remote JWKS JSON | Based on IdP policy, and refresh based on the cache TTL setting. |

# Run the application in production

Running the application in a production environment requires a secure, consistent, and automated approach. Unlike development setups, production deployments must ensure scalability and compliance with security best practices. This includes using hardened configurations, externalizing secrets, and managing container lifecycle effectively.

This topic explains the recommended methods for running the application in a production environment. It provides options for using the helper script, Docker Compose, or a container orchestrator such as Kubernetes.

### Option A: Use the run.sh script (recommended)

The exported production bundle includes a `run.sh` helper script that simplifies deployment and lifecycle management. This script ensures consistent commands and applies environment overrides automatically.

The following code block shows how to use the `run.sh` script on a production host:

```
# On production host with exported bundle
./run.sh start    # Start the server
./run.sh stop     # Stop the server
./run.sh restart  # Restart the server
./run.sh logs     # View logs
```

The `run.sh` script automatically:

- Loads the correct Docker Compose configuration
- Applies environment variable overrides
- Manages container lifecycle
- Provides consistent deployment commands

### Option B: Use Docker Compose directly

You can run the application using Docker Compose commands. The following command starts the application using the exported Docker Compose file:

```
docker compose -f docker-compose.mcp.yml up -d
```

### Option C: Use a container orchestrator (Kubernetes)

For large-scale or cloud deployments, you can convert the Docker Compose configuration into Kubernetes manifests. This approach provides better scalability and orchestration features. The following guidelines apply when deploying with Kubernetes:

- Convert the Docker Compose configuration into a **Deployment** and **Service**.
- Mount configuration files and certificate secrets as read-only volumes.
- Configure a readiness probe that checks `/http/health` if exposed, or run a lightweight tool invocation for health checks.

# Scaling considerations

Scaling in production ensures that the application can handle increased traffic and maintain high availability. Proper configuration and load balancing are critical for secure and efficient scaling.

## Horizontal scaling

Horizontal scaling refers to adding more instances of the application to distribute the workload across multiple servers or containers.

The following points describe the key considerations for horizontal scaling:

- **Stateless design**—Multiple replicas can run safely. Ensure that all replicas use consistent configuration and the same public key.
- **No sticky sessions**—Token exchange caches, if present, are small and ephemeral. This eliminates the need for session affinity.
- **Load balancer readiness**—Place the application behind an L4 or L7 load balancer. Ensure that the TLS termination strategy is consistent, whether using end-to-end encryption or edge termination.

## Vertical scaling

Vertical scaling refers to increasing the resources, such as CPU, memory of a single server or container to handle more load.

The MCP server uses Uvicorn with asyncio for high-performance concurrent request handling. Uvicorn is a lightning-fast Asynchronous Server Gateway Interface (ASGI) server designed for Python applications, whereas asyncio is a Python standard library that provides an event loop for asynchronous programming. By default, the server runs with a single worker, which is suitable for most I/O-bound workloads. For CPU-intensive or mixed workloads, you can increase the number of workers to utilize multiple CPU cores effectively.

### Default configuration: Single worker

The default configuration uses a single worker process. This setup is optimized for typical MCP operations that are I/O-bound and require efficient concurrency handling.

The following settings describe the default behavior when running with a single worker:

- **Default setting**—`UVICORN_WORKERS=1`
- **Best for**—I/O-bound workloads that are typical of MCP operations
- **Concurrency**—Asyncio event loop handles thousands of concurrent connections
- **Memory usage**—Lower memory footprint per connection

### Multi-worker configuration

For production environments, you can configure multiple workers to take advantage of available CPU cores. This approach improves scalability and performance for compute-intensive workloads.

These examples show how to configure multiple workers for production scaling based on CPU count:

```
# Production scaling based on CPU count
export UVICORN_WORKERS=$(nproc)        # Linux: use all CPU cores
export UVICORN_WORKERS=$(sysctl -n hw.ncpu)  # macOS: use all CPU cores

# Conservative approach (recommended)
export UVICORN_WORKERS=$(($(nproc) - 1))     # Leave 1 core for system
```

### Scaling guidelines

Refer to these guidelines for setting worker counts according to workload characteristics:

| Workload type | Worker count | Reasoning |
|---|---|---|
| I/O heavy (typical) | 1–2 workers | Asyncio handles concurrency efficiently |
| CPU intensive | CPU cores minus 1 | Enables parallel processing without overcommit |
| Mixed workload | CPU cores divided by 2 | Balances parallelism and overhead |
| Memory constrained | 1 worker | Each worker loads the full application |

## Environment variables for production tuning

When you export a production bundle, the generated `docker-compose.mcp.yml` file includes default Uvicorn settings. These defaults can be overridden using environment variables to tune worker counts, proxy headers, and other parameters for production environments. For example, this code block shows the default environment variable configuration in the auto-generated `docker-compose.mcp.yml` file:

```
# In docker-compose.mcp.yml (auto-generated)
environment:
  UVICORN_WORKERS: '${UVICORN_WORKERS:-1}'              # Override via .env file
  UVICORN_PROXY_HEADERS: '${UVICORN_PROXY_HEADERS:-0}'  # Set to 1 behind a Load Balancer
# ... other settings
```

Where,

- `UVICORN_WORKERS` controls the number of worker processes. The default is 1.
- `UVICORN_PROXY_HEADERS` should only be enabled when the application runs behind a trusted load balancer.
- Additional settings can be added or overridden using an `.env` file or host environment variables.

## Methods to override environment variables

To optimize performance and security, you can override the default Uvicorn settings in the generated `docker-compose.mcp.yml`. The recommended approach is to use an `.env` file, but other methods are also supported. For production tuning, you can override Uvicorn environment variables in the following ways:

- **Use an `.env` file (recommended)**—Create an `.env` file in the same directory as `docker-compose.mcp.yml` and define the following variables:

  ```
  # Create .env in same directory as docker-compose.mcp.yml
  UVICORN_WORKERS=4                             # Worker processes based on CPU count
  UVICORN_PROXY_HEADERS=1                       # Enable when behind load balancer
  UVICORN_FORWARDED_ALLOW_IPS="10.0.0.0/8"     # Trust specific proxy IPs (security
  best practice)
  ```

- **Use server environment file**—Edit the `server.env` file in the generated profile directory to tune settings for your workload:

  ```
  # Edit generated/<profile>/server/server.env
  UVICORN_WORKERS=2                    # Tune for your workload
  UVICORN_KEEPALIVE=5                 # Keep-alive timeout
  UVICORN_SHUTDOWN_TIMEOUT=0          # Graceful shutdown timeout
  ```

- **Use host environment variables**—Export environment variables on the host before running Docker Compose:

  ```
  # Export before running docker-compose
  export UVICORN_WORKERS=$(nproc)     # Use all CPU cores
  export UVICORN_PROXY_HEADERS=1       # Behind load balancer
  docker-compose up
  ```

### Security considerations for proxy configuration

When deploying the application behind a load balancer or reverse proxy, it is critical to configure proxy-related settings securely. Incorrect configuration can expose the application to header spoofing attacks or allow untrusted sources to manipulate request routing. Uvicorn provides environment variables to control how proxy headers are handled, but these must be set carefully to prevent security vulnerabilities. The following table lists the environment variables you can define in the `.env` file and their purpose:

| Environment variable | Example value | Description |
|---|---|---|
| `UVICORN_WORKERS` | `4` | Safe for direct deployments. Ignores all `X-Forwarded-*` headers. |
| `UVICORN_PROXY_HEADERS` | `1` | Enable only when the application runs behind trusted load balancers or proxies. |
| `UVICORN_FORWARDED_ALLOW_IPS` | `"10.0.0.0/8"` | Always set to specific proxy IP ranges in production. The default trusts all IPs, which is not secure. |

## Observability in production

Observability in a production environment is essential for monitoring application health, diagnosing issues, and ensuring compliance. Proper logging and monitoring practices help you gain insight into system behavior without exposing sensitive data. This section explains how to enable structured logging, ship logs securely, and manage payload logging with caution.

### Enable structured logging

To improve log readability and integration with monitoring tools, enable structured logging in the configuration:

```
observability.logging.json=true
```

### Manage secrets securely

Secrets such as API keys, tokens, and certificates must be handled with strict security controls to prevent exposure. Follow these recommended practices to manage secrets:

- Mount secrets as read-only to prevent accidental modification.
- Set file permissions to `0400` wherever possible to restrict access to the owner only.

### Ship logs securely

Ship logs to a centralized system for analysis. You can use one of the following approaches:

| Method | Description |
|---|---|
| Sidecar container | Deploy a logging sidecar to collect and forward logs |
| Docker logging driver | Use Docker’s built-in logging driver to stream logs to an external service |

### Manage payload logging carefully

Do not enable payload logging unless you have a robust redaction strategy in place. Logging raw payloads can expose sensitive information such as tokens or personally identifiable data.

## Security and performance checklist

Before deploying to production, it is important to validate that the application meets security and performance requirements. This checklist ensures that authentication, authorization, and resource protection measures are properly configured. It also includes performance tuning recommendations for production environments.

### Authentication and authorization requirements

Some of the essential security measures for authentication and authorization include:

| Requirement | Description |
|---|---|
| Service account required | Set `required=true` and enforce issuer and audience validation if applicable |
| Key rotation process | Use JWKS or establish a planned key rotation strategy |
| Scope definition | Define scopes and ensure sensitive tools are not exposed without explicit scope |
| TLS or mTLS enforced | Do not allow plaintext ingress |
| Token exchange security | If enabled, use HTTPS and restrict scopes |
| Route maps | Include only intended write paths |
| Secrets management | Mount secrets externally. Do not bake them into images or configuration files |

### Performance and resource protection for production hardening

The following checklist provides recommended settings for performance tuning and resource protection:

| Setting | Recommended value or action |
|---|---|
| Request timeout | Enable `middleware.request_timeout.enabled=true`. The default value is 500 ms. |
| Payload size guard | Configure `runtime.max_request_bytes=10240`. Adjust this value per use case. |
| Circuit breaker | Enable `middleware.circuit_breaker.enabled=true` and tune thresholds |
| Connection pool limits | Set `runtime.http.limits.max_connections=100` and `max_keepalive_connections=20` |
| Response guard | Enable blocking or trimming with sensible limits |
| Rate limiting | Enable for public or shared deployments |
| Method validation | Enable fast-fail validation: `middleware.method_validation.enabled=true` |
| Uvicorn workers | Configure `UVICORN_WORKERS` based on workload |
| Connection tuning | Optimize `UVICORN_KEEPALIVE` and `UVICORN_SHUTDOWN_TIMEOUT` for load |
| Proxy headers | Enable `UVICORN_PROXY_HEADERS=1` when behind a load balancer |
| Resource monitoring | Track CPU, memory, and response time |

### Routing and exposure

Routing and method exposure settings control which HTTP methods are allowed and how requests are validated. These configurations help enforce security and prevent accidental execution of unsupported operations. The following table summarizes the routing and exposure rules:

| Setting | Description |
|---|---|
| Non-GET methods | All write methods must be explicitly included. None are allowed implicitly. |
| `method_validation` | Enabled by default to reject unknown tools or unsupported methods quickly. |

## Rotate security artifacts in production

Rotation procedures ensure that cryptographic materials and security artifacts remain secure and compliant with organizational policies. Regular rotation reduces the risk of compromise and maintains trust in authentication and encryption mechanisms.

The following table lists the artifacts that require rotation, the triggers for rotation, and the recommended methods:

| Artifact | Rotation trigger | Method |
|---|---|---|
| Service account public or private key | Periodic rotation every 90–180 days or upon compromise | Generate a new key pair, deploy the public key, issue new tokens, and retire the old key |
| Service account JWT | Token expiry or scope change | Use `remint-all` or an external signer pipeline |
| TLS certificate | Certificate expiry based on CA policy | Automate renewal using ACME or corporate PKI |
| JWKS keys | As per identity provider rotation schedule | Rely on JWKS cache TTL and monitor for failures |

# Disaster recovery guidelines for MCP servers

Disaster recovery planning ensures that the system can be restored quickly after a failure or outage. MCP servers are designed to be stateless, which simplifies recovery and rollback procedures. Some of the disaster recovery best practices include:

- Configuration and public artifacts can reconstitute the server quickly because there is no mutable database state.
- Keep export directories, excluding secrets, in version control or artifact storage to enable fast rollbacks.

# Common failure modes and resolutions

Understanding common failure modes helps you diagnose issues quickly and apply the correct resolution steps. The following table lists symptoms, their impact, and recommended resolutions:

| Symptom | Impact | Resolution |
|---|---|---|
| JWKS endpoint timeout | Service account authentication failures (401 errors) | Provide a fallback static key or restore identity provider availability |
| Scope misconfiguration hides all tools | Empty tool catalog | Adjust default scope mapping |
| Response guard blocked key tool | Tool becomes unusable | Increase the limit, test the configuration, and redeploy |

# 15

# Rate limiting in OpenEdge MCP Server

Rate limiting is a mechanism that controls the number of requests a client or all clients can make within a specified time interval. It ensures that the system operates efficiently and prevents excessive or abusive request patterns.

The following table summarizes the goals of implementing rate limiting:

| Goal | Description |
|---|---|
| Prevent excessive requests | Avoid abusive or accidental high-frequency tool invocations. |
| Ensure fairness | Provide per-client fairness by default and allow a simpler global cap when required. |
| Enable auditing | Emit structured audit events for both allow and deny decisions. |
| Support development flexibility | Allow rate limiting to be easily disabled during early development. |

For details, see the following topics:

- Implement rate limiting
- Configuration keys for rate limiting
- Recommended rate limiting values for different environments
- Monitor rate limiting through observability and audit fields
- Safe rollout strategy for rate limiting
- Alerting suggestions for rate limiting

- Tune and troubleshoot rate limiting

# Implement rate limiting

This topic explains the implementation details of rate limiting in the MCP Server.

When the `runtime.middleware.rate_limiting.enabled` property is set to `true`, the system adds a rate limiting middleware to enforce request quotas. An auditing layer logs each allow or deny decision under the category `rate_limit`.

The following table lists the main design principles:

| Design point | Description |
|---|---|
| Enforcement method | Token bucket style enforcement supports requests per second (RPS) and optional burst capacity. |
| Per-client mode | Uses a derived key hierarchy: `client_id` → `session_id` → `global fallback`. |
| Global mode | Uses a single shared bucket for all requests. |
| Fail-fast behavior | Excess requests trigger an MCP error and the client receives a structured denial response. |

# Configuration keys for rate limiting

Configuration keys are specific properties or settings that control how a feature behaves. These keys are defined in the configuration file of the server, for example `runtime.middleware.rate_limiting`, and allow administrators or developers to enable, disable, or fine-tune a functionality.

The following table lists the configuration keys under `runtime.middleware.rate_limiting`:

| Key | Data type | Default | Description |
|---|---|---|---|
| `enabled` | Boolean | `false` | Enables or disables rate limiting. This is the master switch. |
| `max_requests_per_second` | Float | `10.0` | Specifies the steady-state permitted request rate. Fractional values are allowed, for example, `2.5`. |
| `burst_capacity` | Integer | *(unset)* | Defines the number of extra tokens that allow short spikes above the `max_requests_per_second` value. If unset, a minimal burst capacity (one token) applies. |
| `global` | Boolean | `false` | Determines the scope of rate limiting. If `true`, all clients share a single bucket. If `false`, each client has its own bucket. |

The following example shows how to enable rate limiting and configure its properties:

```
{
   "runtime": {
    "middleware": {
      "rate_limiting": {
        "enabled": true,
        "max_requests_per_second": 5,
        "burst_capacity": 15,
        "global": false
      }
    }
```

# Recommended rate limiting values for different environments

You can select suitable values for rate limiting properties based on different usage scenarios. The goal should be to balance system performance, fairness, and protection against overload.

The following table lists suggested configurations for common environments:

| Scenario | Suggested `max_requests_per_second` | Suggested `burst_capacity` | Global mode? | Rationale |
|---|---|---|---|---|
| Local development | 20 | 40 | `false` | Use generous limits to allow rapid iteration during development. |
| Public demo | 2 | 6 | `true` | Apply simple global protection to prevent abuse during demonstrations. |
| Production (agents) | 1-3 | Three times the steady-state rate | `false` | Limit sustained load per client to maintain system stability. |

# Monitor rate limiting through observability and audit fields

The OpenEdge MCP Server records audit events for every request that is either allowed or denied by the rate limiting mechanism. These events help administrators monitor system behavior and troubleshoot issues.

The following table lists the fields included in each audit event:

| Field | Description |
|---|---|
| `ts` | Specifies the timestamp of the event in UTC format. |
| `category` | Specifies the category of the event. For rate limiting, this value is `rate_limit`. |
| `action` | Specifies the action performed. For rate limiting checks, this value is `check`. |
| `status` | Indicates whether the request was allowed or denied. |
| `key` | Specifies the subject key used for rate limiting. This can represent a client, a session, or a global bucket. |
| `reason` | Provides the reason for denial when applicable. For example, `rate_limit`. |
| `client_id` or `session_id` | Specifies the identifiers for the client or session, if available from the FastMCP context. |

Use structured logging to ensure audit events can be parsed consistently and integrated with monitoring tools.

## Safe rollout strategy for rate limiting

You can use the following approach to introduce rate limiting safely into your environment:

1. Start with disabling rate limiting by setting `enabled=false`.
2. Enable rate limiting with generous values such as `max_requests_per_second=20` and `burst_capacity=40`. Monitor logs for behavior.
3. Adjust to target steady-state values after observing telemetry for 24 hours.
4. Add alerting mechanisms for thresholds such as a high number of denies per minute.

## Alerting suggestions for rate limiting

The following table outlines the conditions that should trigger alerts to help detect misconfigurations or abusive workloads:

| Condition | Recommended action |
|---|---|
| Deny rate exceeds 5% of total requests | Investigate for abusive workloads or incorrect burst configuration. |
| Sudden drop in throughput with rising deny count | Check for accidental loops or broken client retry logic. |

## Tune and troubleshoot rate limiting

This topic explains common symptoms related to rate limiting, their possible causes, and recommended adjustments. Use this information to fine-tune configurations and resolve issues effectively.

The following table lists typical issues and how to address them:

| Symptom | Possible cause | Recommended adjustment |
|---|---|---|
| Frequent denies early in bursts | `burst_capacity` is too low | Increase `burst_capacity` to 2–3 times the `max_requests_per_second` value. |
| Spiky latency even without denies | Bucket depletion combined with jitter | Lower `max_requests_per_second` if upstream systems are overloaded, or add downstream caching to smooth traffic. |
| All clients throttled simultaneously | `global` mode enabled unintentionally | Set `global=false` to ensure per-client fairness. |
| Single runaway client unthrottled | Client IDs not identified correctly | Ensure each client sets a distinct `client_id`. Verify this using audit logs. |

# 16

# Upgrade process and key or certificate rotation

This topic describes safe upgrade steps for the OpenEdge MCP Server and operational procedures for rotating service account keys, JWKS-backed keys, and TLS or mTLS certificates.

The following objectives guide the upgrade and rotation process:

| Objective | Description |
|---|---|
| Minimize downtime | Ensure that upgrades do not disrupt service availability. |
| Preserve deterministic profiles | Maintain configuration provenance for consistent deployments. |
| Enforce cryptographic hygiene | Use short-lived tokens and perform periodic key rollover to maintain security. |

For details, see the following topics:

- Version and upgrade principles
- Pre-upgrade checklist and upgrade procedure
- Service account key rotation
- TLS and mTLS certificate rotation
- Token renewal and revocation
- Patterns to minimize downtime
- Observability hooks during rotation

- Rollback decision matrix
- Automation suggestions

# Version and upgrade principles

The following table outlines the principles that govern versioning and upgrade processes:

| Principle | Explanation |
|---|---|
| Reproducibility | Always regenerate or export from a tagged version such as a Git tag or image digest. |
| Immutable runtime | Production environments must run only exported bundles with pinned image digests. |
| Fast rollback | Keep the previous exported bundle intact so that rollback involves bringing down the current compose and bringing up the previous compose. |

# Pre-upgrade checklist and upgrade procedure

Before performing an upgrade, use the following checklist to ensure readiness:

| Item | Description |
|---|---|
| Retain exported bundle | Verify that the exported bundle for the current version is retained in the `exports/<name>-<oldver>` directory. |
| Review OpenAPI specification changes | Confirm that the OpenAPI specification differences are reviewed. Ensure that `operationIds` remain stable and any changes are intentional. |
| Validate configuration changes | Validate changes in `mcp_server_config.json` in the staging environment. Confirm that response guard and rate limiting settings are still appropriate. |
| Run diagnostic checks | Ensure that the new version passes the `./mcpgen doctor` command successfully. |
| Document new configuration keys | Verify that any new configuration keys are documented and that their default values are acceptable. |

### Upgrade procedure from development to production

Follow these steps to upgrade the OpenEdge MCP Server from development to production:

1. Pull the new Git tag or image digests.
2. Regenerate the profile in development by using the same specification and prompts to regenerate the profile:

```
./mcpgen up --no-prompt --tls --spec ./openapi.yml
```

3. Perform smoke tests and run representative tool invocations.
4. Export the production-ready bundle using the following command:

```
./mcpgen export prof --bundle prod
```

5. Diff the exported `mcp_server_config.json` against the previous version. Confirm that only intended changes are present.
6. Deploy the new bundle alongside the old bundle on a different port. Validate functionality in staging.
7. Perform production cutover by scaling down the old compose after the health check passes on the new deployment.
8. Monitor logs for errors, rate limit issues, and response guard events for 15 to 30 minutes.
9. If a regression occurs, stop the new bundle and restart the previous bundle without changes.

## Service account key rotation

This topic describes recommendations for service account (SA) key rotation and provides steps for rotating static `Ed25519` keys and JWKS-backed keys.

### Recommendations for key rotation

The following table summarizes key rotation aspects and recommended practices:

| Aspect | Recommendation |
|---|---|
| Key type | Use `Ed25519`, which is the default or `RS256` when integrating with JWKS. |
| Rotation interval | Rotate keys every 90 days in development environments and every 30 to 60 days in production environments that handle sensitive data. |
| Overlap | Keep the previous public key available until all tokens minted with it have expired. |

### Rotate static `Ed25519` keys

Follow these steps to rotate static `Ed25519` keys:

1. Use `mcpgen` or external tooling to generate a new key pair.
2. Update the profile or the exported bundle secret source with the new private key.
3. If token verification is externalized, for example downstream auditing, keep the old public key available until all tokens signed with it expire.
4. Generate new SA tokens using the updated key.
5. Securely delete the old private key after all tokens using it have expired.

### Rotate JWKS-backed keys

JWKS rotation is managed by the identity provider (IdP). The OpenEdge MCP Server consumes new keys when the JWKS cache expires. After configuring JWKS rotation, verify these settings to maintain reliable key updates:

- Ensure that the `jwks_cache_seconds` setting is configured to a small but reasonable value, such as 300 to 900 seconds, to pick up rotations promptly.
- Monitor for verification failures, such as bursts of HTTP 401 errors, which may indicate premature removal of old keys.

## TLS and mTLS certificate rotation

This topic describes recommended certificate lifetimes for development and production environments and provides steps for rotating TLS and mTLS certificates. It also explains how to manage certificate hostnames during regeneration.

### Recommended certificate lifetimes

The following table summarizes typical certificate lifetimes and production recommendations:

| Certificate type | Typical lifetime (development) | Production recommendation |
|---|---|---|
| Self-signed development certificate | 365 days | Use shorter lifetimes (90–180 days) or switch to an internal CA or ACME-based solution |
| Client mTLS certificate | 180 days | Rotate every 60–90 days with automation |
| Certificate authority (development profile) | 1 year | Use a managed CA service or internal PKI |

## Steps to rotate server TLS certificates

Perform the following steps to rotate TLS certificates:

1. Use ACME, an internal PKI, or a development script to generate the new certificate and key.
2. Place the certificate and key in the correct location:
   - For development, place them in the `profile/certs/` directory.
   - For production, use a secure secret mount or a vault reference if externalized.
3. Ensure that the configuration reflects the new certificate and key filenames.
4. Perform a rolling restart using the following command:

```
docker compose up -d
```

5. Distribute new client certificates and maintain the CA trust chain until all clients are updated.

## Manage certificate hostnames

When regenerating certificates with different hostnames or domains, follow these guidelines:

- **Development environment using `mcpgen`**—Use the `--extra-hostnames` flag to include additional domains when generating certificates:

```
./mcpgen up --quick --tls --extra-hostnames api.example.com,staging.example.com
```

  Generated certificates include both `localhost` and the specified hostnames as Subject Alternative Names (SAN). Both `.pem` and `.crt` formats are generated for client compatibility.

- **Manual certificate generation**—Use the same `--extra-hostnames` flag for manual generation:

```
./mcpgen up --quick --tls --extra-hostnames "api.example.com,staging.example.com
```

  For mTLS, include client certificates:

```
./mcpgen up --quick --mtls --extra-hostnames "api.example.com,staging.example.com"
```

## Production considerations

Use the following guidelines when managing certificate hostnames in a production environment:

- Plan hostname changes during scheduled maintenance windows.
- Update DNS records before deploying new certificates.
- Verify that all client applications can validate the new certificate hostnames.
- Consider using wildcard certificates (for example, `*.example.com`) for flexibility across subdomains.

# Token renewal and revocation

Managing the renewal and revocation of JSON Web Tokens (JWT) for service accounts and exchanged user tokens is important because short-lived tokens and frequent key rotation help protect your system from security breaches. By limiting token lifetime and rotating keys quickly, you reduce the risk of unauthorized access if a token is leaked.

## Recommended renewal and revocation strategies

Use the following table to understand renewal triggers and revocation strategies for different token types:

| Token type | Renewal trigger | Revocation strategy |
|---|---|---|
| Service account JWT | Renew before expiration, for example at 80% of its lifetime | Rotate the signing key. Old tokens will naturally expire because of their short time-to-live (TTL). |
| Exchanged user token | Renew per session or per API call, or use a short TTL of 5 to 15 minutes | Rely on the upstream identity provider (IdP) for revocation and blacklist enforcement. |

**Note:** Short-lived tokens combined with fast key rotation significantly reduce the blast radius in case of token leakage.

# Patterns to minimize downtime

Minimizing interruptions ensures that clients and applications continue to function without noticeable delays. The following table describes common patterns that help reduce downtime during upgrades or deployments. Each pattern includes a description and its trade-offs:

| Pattern | Description | Trade-off |
|---|---|---|
| Side-by-side compose | Start a new stack on an alternate port and then switch the client to the new stack | Requires dynamic endpoint switching or a load balancer |
| Blue or green DNS | Maintain two DNS targets and flip traffic after health checks pass | Requires tuning DNS TTL values for fast propagation |
| In-place rolling | Stop the old stack and start the new stack quickly | Causes brief unavailability unless a load balancer manages multiple instances |

# Observability hooks during rotation

You need to monitor system health during key and certificate rotation because these changes can introduce authentication failures, TLS handshake issues, or middleware disruptions. Observability hooks help you detect problems early and confirm that critical security and performance features remain intact after an upgrade.

Use the following checks to ensure system stability during rotation:

- Watch for a sudden spike in authentication error rates during key swaps.
- Track TLS handshake failures after certificate rotation.
- Confirm that rate limit audits continue to log events. This approach ensures that middleware remains intact after an upgrade.

# Rollback decision matrix

When issues occur during or after rotation, you must decide quickly whether to roll back. The rollback decision matrix provides clear guidance on what action to take based on specific symptoms and ensures fast recovery and minimized downtime, and prevents prolonged service interruptions.

Use this matrix to determine the appropriate rollback action based on observed symptoms:

| Symptom | Action |
|---|---|
| Surge in HTTP 401 errors after service account key swap | Restore the previous key and reissue tokens |
| TLS handshake errors for some clients | Verify the intermediate CA chain. Revert to the prior certificate if the issue is urgent |
| Increased latency with no code changes | Check token exchange and JWKS network resolution. Revert if the issue persists |
| Missing operations in tool discovery | Compare the OpenAPI specification differences. Roll back if there is a specification regression |

# Automation suggestions

Manual rotation and validation are error-prone and time-consuming. Automation ensures consistency and enforces security best practices. These suggestions help you schedule regular checks, validate endpoints, and rotate keys without human intervention.

Consider implementing the following automation tasks to improve reliability and reduce manual effort:

- **Continuous integration job**—Regenerate the profile weekly using the `--no-prompt --no-sa` flags to test specification parsing and build integrity.
- **Scheduled job**—Validate the JWKS endpoint using a `HEAD` request and cache purge. Report any unreachable states.
- **Cron or pipeline task**—Rotate development SA keys automatically every month.

# VI

# OpenAPI specification and advanced configuration

For details, see the following topics:

- OpenAPI specification quality guidelines

# 17

# OpenAPI specification quality guidelines

This section provides guidelines on how to create high-quality OpenAPI specifications for use with the OpenEdge MCP Server. Following these guidelines ensures that the MCP tools generated from your specifications are clean, intuitive, and predictable. High-quality specifications provide the following benefits:

- Clear tool names and arguments derived from `operationId`, parameters, and request body schemas
- Accurate input validation possibilities
- Helpful tool descriptions that improve agent reasoning
- Reduced ambiguity in downstream integration

For details, see the following topics:


- Core principles for OpenAPI specifications
- Request body guidelines
- Define response schemas
- Component schema hygiene
- Define titles and descriptions for schemas and operations
- Ensure clarity in parameter definitions
- Error conventions
- Versioning and deprecation guidelines
- Hand-editing checklist before commit
- Common pitfalls in API specifications

- Minimal example for API specification

# Core principles for OpenAPI specifications

When designing OpenAPI specifications for MCP tools, it is important to follow certain principles that ensure clarity and consistency while ensuring usability. These principles help create predictable tool behavior, improve integration, and make the specification easier to maintain. The following topic describes the fundamental principles that help you design robust and maintainable OpenAPI specifications.

The following table lists the key principles, explains why they matter, and provides examples:

| Principle | Why it matters | Example |
|---|---|---|
| Unique `operationId` per operation | MCP tool IDs map one-to-one to `operationId` | `getCustomerById`, `listCustomers` |
| Descriptive summary and rich description | Tool list shows summary; description becomes long help | Summary: *Fetch a customer*; Description: business rules, rate limits |
| Explicit request and response schemas | Enables clear input/output contracts | `requestBody → $ref: '#/components/schemas/CreateOrderRequest'` |
| Use `title` in component schemas | Provides readable names in generated helper tools | `title: Customer` |
| Every object property has a `type` | Avoids defaulting to untyped `any` | `type: string` or `type: integer` |
| Mark required fields using `required` array | Distinguishes mandatory versus optional fields | `required: [id, name]` |
| Enumerate finite sets with `enum` and description | Lets agents constrain values | `status: {enum: [OPEN, CLOSED]}` |
| Explain side effects in description | Tool consumers can reason about writes | *Creates invoice; idempotent by clientProvidedId* |
| Version the specification (`info.version`) | Provides traceability and cache invalidation | `1.3.0` |
| Avoid generic names | Improves clarity of tool list | Prefer `getOrderInvoices` over `doList` |

# Request body guidelines

Request bodies define the structure and content of data sent to an API operation. Properly specifying request bodies ensures accurate input validation, clear contracts, and better reuse of schemas. This section provides best practices for defining request bodies in your OpenAPI specification.

### Key recommendations

The following recommendations help you define request bodies clearly and consistently in your OpenAPI specification:

- Always specify `content` as `application/json` (or other media types) with a schema reference.
- Use a top-level object schema instead of inline anonymous structures. This approach allows reuse and better naming.
- Indicate optional versus required fields using the `required` array. Do not rely solely on description text.
- If using arrays, fully specify the `items` schema, including `type`, `$ref`, and `description`.
- Provide field-level `description` for non-obvious semantics or constraints.

### Example of a well-defined request body

The following example demonstrates how to define a request body and responses for creating a new customer:

```
post:
  summary: Create a new customer
  description: Creates a customer record. Fails if externalId already exists.
  operationId: createCustomer
  requestBody:
    required: true
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/CreateCustomerRequest'
  responses:
    '201':
      description: Customer created
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Customer'
    '409':
      description: Conflict – externalId already exists
```

## Define response schemas

Response schemas are structured definitions that describe the format and content of data returned by an API. They ensure that both the API provider and the consumer have a clear, predictable understanding of what the response should look like.

To define clear response schemas, follow these guidelines:

- Always define a primary success code for each response. Common examples include:

| Status code | Description |
|---|---|
| 200 | Successful request |
| 201 | Resource created |

- Define an explicit error schema such as `ErrorResponse` to ensure predictable error handling. This schema should include fields that describe the error in detail.
- Avoid unnecessary complexity in schema design. Use `oneOf`, `allOf`, or `anyOf` only when strictly required. Prefer simpler schema shapes for better readability and maintainability.
- For list endpoints, use an envelope that includes metadata along with the items. This approach provides additional context such as total count and pagination details. Here is an example schema for a customer list:

```
CustomerList:
  title: CustomerList
  type: object
  required: [items, total]
  properties:
    items:
      type: array
      items:
        $ref: '#/components/schemas/Customer'
    total:
      type: integer
      description: Total available (not just page size)
    nextCursor:
      type: string
      nullable: true
```

- Use format attributes such as `date-time`, `uuid`, or `email` to provide additional validation and clarity.

# Component schema hygiene

Component schema hygiene is the practice of defining schemas in a structured, explicit, and reusable way. Poor schema hygiene can lead to ambiguous documentation and unpredictable tooling behavior, which leads to reduced maintainability. This topic explains best practices for maintaining clean and consistent component schemas in API specifications.

### Common anti-patterns and preferred approaches

The following table lists common anti-patterns, their preferred alternatives, and the rationale for adopting best practices:

| Anti-pattern | Preferred approach | Rationale |
|---|---|---|
| Implicit object `{}` with no type | Use `type: object` with `properties` and `title` | Improves clarity and supports downstream code generation |
| Missing property types | Define explicit types such as `type: string` | Prevents fallback to generic types |
| Nested anonymous objects | Extract to a named component with a `title` | Promotes reuse and cleaner documentation |
| Overuse of `additionalProperties: true` | Use explicit properties or constrained maps | Avoids “anything goes” behavior in tooling |
| No examples | Add `example:` or `examples:` with realistic payloads | Helps agents and developers understand usage |

### Provide examples for complex objects

Always include at least one of the following for complex objects:

- `example`
- `examples`
- `x-*` extension with realistic payload snippets

---

**Note:** Examples improve reasoning for automated agents and help developers understand expected data structures.

---

## Define titles and descriptions for schemas and operations

Effective titles and descriptions for API components and operations provide guidance on summarizing functionality, detailing business rules, and including technical considerations such as authorization and idempotency.

### Titles and descriptions for component schemas

Component schemas require clear and concise titles and descriptions to help developers understand their purpose and usage. Here are the recommeded guidelines:

- For `title`, use a short, descriptive phrase that identifies the schema.
- Begin the `description` with a concise sentence that summarizes the schema. Follow with detailed rules and constraints.

### Summaries and descriptions for operations

Operations need summaries and descriptions that convey intent and technical details. Summaries should be brief, while descriptions can include multiple paragraphs. Recommended guidelines include:

- Limit `summary` to 60 characters. State the purpose of the operation clearly.
- For `description`, provide detailed information, including:
  - **Business rule edge cases**—Describe any exceptions or special conditions that apply to the operation.
  - **Authorization expectations**—Specify the required scopes or roles that are necessary to perform the operation.
  - **Idempotency guarantees**—Indicate whether the operation is idempotent and explain the conditions under which this guarantee holds.
  - **Rate limit nuances**—Provide information about any rate-limiting policies or thresholds that affect the operation.
  - **Side effect notes**—Explain any downstream workflows or triggers that occur as a result of executing the operation.

## Ensure clarity in parameter definitions

Clear and consistent parameter definitions improve usability and reduce ambiguity for developers. The table below summarizes the aspects of parameter definitions and the corresponding guidelines:

| Parameter aspect | Guideline |
|---|---|
| Names | Use consistent names across all endpoints. For example, use `customerId` instead of alternating between `id` or `custId`. |
| Path parameters | Always declare `required: true` and specify `schema.type` for each path parameter. |
| Query pagination | Standardize pagination by using either `limit` and `cursor` or `page` and `pageSize`. Document the chosen approach in the parameter description. |
| Booleans | Avoid tri-state confusion. If a boolean parameter is nullable, document its semantics clearly. |
| Filtering arrays | Use repeatable query parameters, for example `filter=val1&filter=val2`, or comma-separated values (CSV). Document which approach is implemented. |

If pagination and filtering are widely used across the API, include a short section in the global description (information section) to explain the standard approach. This ensures consistency and helps developers understand the expected patterns.

# Error conventions

Error conventions define a consistent approach for representing and handling errors in API responses. They ensure that clients receive predictable and structured error information that improves debugging and automation and enhances the overall user experience. You can specify how errors should be structured, what fields should be included, and how these schemas should be referenced in non-successful responses.

### Define a canonical error schema

A canonical error schema serves as the single source of truth for error representation. It should include essential fields such as an error code, a message, and optional details for context.

The following example shows a recommended error schema:

```
ErrorResponse:
  title: ErrorResponse
  type: object
  required: [code, message]
  properties:
    code:
      type: string
      description: Machine readable error code
    message:
      type: string
      description: Human readable message
    details:
      type: object
      additionalProperties: true
      description: Optional structured context
```

Always reference the canonical error schema in all non-successful responses, such as `4xx` and `5xx` status codes. This approach ensures that generated tools and client applications can handle errors consistently.

# Versioning and deprecation guidelines

To manage API versioning and handling deprecated features in a way that ensures clarity and minimizes disruption for consumers, follow these guidelines that help maintain backward compatibility and provides transparency during major changes:

- **Increment the version for breaking changes**—Always update the `info.version` field whenever a breaking change is introduced. This practice signals to consumers that the API has changed in a way that may require updates on their side.
- **Mark deprecated operations and properties explicitly**—Instead of silently removing operations or properties, use the `deprecated: true` attribute. This approach provides clear communication to API consumers and allows them time to migrate before a major version bump.
- **Add change notes for significant breaking changes**—Include a clear change note in the top-level `info.description` field whenever there is a significant breaking change. This approach ensures that consumers are aware of the impact and can plan accordingly.

# Hand-editing checklist before commit

The hand-editing checklist helps maintain high-quality API documentation by enforcing standards for operations, schemas, and error handling. It reduces the risk of incomplete or ambiguous specifications.

The following table lists the key items to review before committing changes:

- Every operation includes a `summary`, `description`, and an `operationId` for clarity and identification.
- Each `requestBody` references a component schema instead of using large inline objects.
- Each success response references a component schema or a primitive schema to maintain consistency.
- Every object schema includes a `title`, specifies `type: object`, and provides a `description` for better documentation.
- All required fields are explicitly listed to prevent ambiguity.
- Property types are defined, and `format` is specified where applicable.
- Enumerations include a description that explains the purpose of each value.
- Examples are added for complex or critical objects to aid understanding.
- Deprecated items are marked using `deprecated: true` and are not removed silently.
- The error schema is referenced in all error responses to ensure consistent error handling.

# Common pitfalls in API specifications

Common pitfalls can lead to tool generation failures, poor documentation quality, and unexpected runtime behavior. Addressing these issues early improves developer experience and reduces maintenance overhead. This topic describes frequent mistakes that occur during API design and their impact on MCP tools. It also provides recommended fixes to avoid these issues.

The following table lists common pitfalls, their impact on MCP tools, and recommended fixes:

| Pitfall | Impact of MCP tools | Recommended fix |
|---|---|---|
| Missing `operationId` | Tool generation fails or produces generic IDs | Add a unique, verb-object style `operationId` |
| Verbose inline request bodies | Harder reuse and cluttered tool descriptions | Extract the request body into a component under `components.schemas` |
| Unbounded arrays | Response guard might truncate unexpectedly | Document the maximum size or introduce pagination |
| Ambiguous numeric types | Agents misinterpret currency versus count | Use `format: int32` or `format: float` and mention the currency unit |
| Empty descriptions | Poor agent reasoning and lack of context | Fill in summaries and descriptions for clarity |

# Minimal example for API specification

This topic provides a minimal yet complete example of an OpenAPI specification that follows best practices. It demonstrates how to structure endpoints, define schemas, and include essential metadata for clarity and consistency.

```
openapi: 3.0.3
info:
  title: Sample Customer API
  version: 1.2.0
  description: >
    Customer management endpoints. Pagination uses limit+cursor. Errors conform to
ErrorResponse.
paths:
  /customers/{customerId}:
    get:
      summary: Get customer
      description: Returns a single customer by id. Requires customer.read scope.
      operationId: getCustomer
      parameters:
        - name: customerId
          in: path
          required: true
          schema:
            type: string
          description: UUID of the customer
      responses:
        '200':
          description: Customer found
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Customer'
        '404':
          description: Not found
  /customers:
    post:
      summary: Create customer
      description: Creates a customer. Idempotent by externalId.
      operationId: createCustomer
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/CreateCustomerRequest'
      responses:
        '201':
          description: Created
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Customer'
components:
  schemas:
    Customer:
      title: Customer
      type: object
      required: [id, name]
      properties:
        id:
            type: string
            format: uuid
        name:
            type: string
        status:
            type: string
            enum: [ACTIVE, INACTIVE]
            description: Current lifecycle status
    CreateCustomerRequest:
      title: CreateCustomerRequest
      type: object
      required: [externalId, name]
      properties:
        externalId:
```

```
          type: string
        name:
          type: string
        status:
          type: string
          enum: [ACTIVE, INACTIVE]
    ErrorResponse:
      title: ErrorResponse
      type: object
      required: [code, message]
      properties:
        code:
          type: string
        message:
          type: string
        details:
          type: object
          additionalProperties: true
```