FTI-Compatible Media Context Protocol Connector

Technical & Functional Documentation

Author : Aman Singh
Supervisor : Jurgen Heyman, Judith Derycke, Yannick De Backer
Institution : **Karel de Grote Hogeschool, Antwerp (Belgium)**

Internship Host : Flanders Technology & Innovation
Repository : https://github.com/Flanders-Technology-Innovation/internship-MCP

Table of Content

Executive Summary (non-technical)

The FTI-Compatible Media Context Protocol Connector is a secure, intelligent gateway that enables organisations to **share digital files and insights safely**—while always keeping **full control over their own data**.

In an increasingly connected world, organisations often need to exchange media files, documents, or AI-generated insights across different systems or partners. But this raises important questions:

- Who controls the data?
- Is it protected in transit?
- Can we trust how it's used or interpreted?

Our connector answers these concerns directly.

It acts as a bridge between internal IT systems and external participants, allowing data exchange under strict conditions defined by the organisation itself. This ensures that nothing is shared accidentally or without explicit permission.

Two Modes for Different Needs

The connector supports two operating modes, each designed for different levels of assurance:

- **Standard Mode** – For everyday interoperability. This mode allows smooth data exchange using standard protocols with other tools and partners that support the Media Context Protocol (MCP).
- **FTI Mode** – For high-security environments. This mode adds stronger safeguards, aligned with the **Flanders Technology & Innovation (FTI)** security profile, suitable for sensitive or regulated data exchanges.

Organisations can choose the mode that fits their current needs—and switch when necessary.

Why This Connector Stands Out

Here’s what makes it especially powerful and relevant:

- **Full Data Control (Data Sovereignty):**
  The organisation remains in charge. No data is shared without a clearly approved and signed request. This guarantees transparency and trust in every exchange.

- **Tailored Access Per User:**
  Every external party uses a **dedicated access key** with specific limits and permissions. This allows fine-grained control over who can request what—and how often.

- **Secure by Design (FTI Mode):**
  When using FTI Mode, all shared files are encrypted with the recipient’s own security keys. Only they can open the data, even if intercepted along the way.

- **Optional AI Tools with Safeguards:**
  The connector also supports AI-powered assistants to help with tasks like reading documents, summarising content, or pulling out useful details. But these features are optional—and always subject to the same strict security rules.

In Summary

This connector brings together **open standards** and **practical security**, allowing organisations to benefit from smarter data sharing—without compromising privacy, control, or trust.

Whether it's used to collaborate with partners, comply with public sector security requirements, or add AI to document workflows, the FTI-Compatible MCP Connector is built to handle it safely, transparently, and efficiently.

Introduction

Modern organisations increasingly rely on the ability to **share data across departments, companies, and systems**—but they must do so in ways that are **secure, controlled, and trustworthy**. This document introduces a solution that helps make that possible.

Background: Why This Matters

At the European and international level, the **International Data Spaces Association (IDSA)** plays a leading role in shaping how data can be shared **responsibly and securely**. Their mission is to promote **data sovereignty**: the principle that every organisation must retain full control over its own data, even as it flows between partners.

In this ecosystem, a special role is played by software components called **connectors**. Think of them as **smart gateways**: they sit between an organisation's internal systems and the outside world, and they manage the flow of data in or out. But they don't just pass data along. Each connector enforces **the rules set by the organisation**—for example, who can access what, under which conditions, and using which formats or technologies.

The **Media Context Protocol (MCP)** builds on this foundation. It is designed specifically for **media-rich data exchange**, where files, documents, or AI-processed content must be handled with structure and precision. MCP defines how digital resources should be:

- **Listed** (so others can find what's available),
- **Described** (so others know what it is and whether they're allowed to access it), and
- **Fetched** (in a controlled, auditable way).

In addition, MCP includes mechanisms for **event-driven updates**—for example, to notify someone when a file becomes available or changes.

This project delivers a connector that speaks MCP natively, **with a special enhancement for high-assurance use cases** aligned with **Flanders Technology & Innovation (FTI)**.

Purpose & Scope of This Project

The purpose of this project is to create a **plug-and-play connector** that:

- Works out-of-the-box with any system or partner already using the **Media Context Protocol**, and

- Adds an **enhanced security mode**, tailored for **sensitive or regulated environments**, such as government or critical infrastructure—where **additional guarantees** around encryption, access control, and auditability are essential.

This documentation is written for **two main audiences**, each with different needs:

- **Decision makers & system integrators**:
  You'll gain a clear understanding of **what this connector does**, how it ensures **data security and sovereignty**, and **why** it might fit into your broader data-sharing or digital transformation strategy.

- **Developers & technical operators**:
  You'll find detailed instructions on **how to install, configure, and maintain the connector** in real-world environments. You'll also learn how to **customise or extend it** to suit your specific needs.

What's Not Included (Yet)

There are a few advanced topics that are considered out of scope for this initial release:

- **Certificate lifecycle management** (e.g. automatic renewal, revocation)

- **Federation between multiple connectors** (e.g. peer discovery, trust negotiation)

These may be included in future updates.

How to Read This Document

- If you need a quick pitch, skim the **Executive Summary** and **Core Concepts**. Depending on your role or interest, you may choose to read different parts of this document first.
- **Just need the essentials?**
  Start with the **Executive Summary** and the **Core Concepts** section. These give a high-level overview of what the connector does and why it's useful.

- **Deploying or managing the system?**
  Head to the **Installation & Setup** and **Deployment & Operations** sections. These will walk you through the practical steps of running the connector in your own environment.

- **Want to understand how it works under the hood?**
  The **Implementation Details** section, along with the **Appendix: API Reference**,

provides a deep technical dive for developers who want to customise or integrate the connector with other systems.

Throughout the document, we've included **illustrations, examples, and architectural diagrams** to help you follow along—whether you're building a proof-of-concept or preparing for full-scale production deployment.

Core Concepts

*Before we explore how the connector works in practice, it's important to establish a **shared understanding of the ideas and vocabulary** that guide its design. Whether you're a system architect, developer, security officer, or business stakeholder, these core concepts will help you see **the big picture** behind sovereign data exchange—and understand why this connector matters.*

*This section lays the groundwork for everything that follows. It explains the **paradigm shift** brought by data spaces and introduces the principles of the **IDSA architecture**, upon which this connector is based.*

IDSA & the Data-Space Paradigm

In today's digital world, sharing data is easy—but **sharing it responsibly** is hard.

The **International Data Spaces Association (IDSA)** is an international consortium that addresses this challenge. Their mission is to enable **trusted, secure, and sovereign data exchange**—especially in environments where different companies, organisations, or public bodies must collaborate without compromising their own data protection standards.

What is Data Sovereignty?

Data sovereignty means that an organisation **never loses control over its data**, even when it is shared. It decides:

- **Who** can access the data,
- **What** they're allowed to do with it,
- **For how long**, and

- **Under what legal or contractual terms**.

This is the heart of the **data-space model**.

How Do Data Spaces Work in Practice?

Imagine an international shipping network.
Each country has a **customs office**. Goods (like food, machinery, or medicines) don't leave the country freely. They go through customs, get inspected, and are stamped with a **manifest** that says what the goods are, who they're for, and what rules must be followed.

Now replace the "goods" with **data**, and you get the idea.

In an IDSA data space:

- Each participant runs a **connector**—like a customs office for data.
- This connector **sits at the edge** of their internal systems.
- All incoming or outgoing data flows through the connector—and only through the connector.
- Internal systems (e.g., databases, APIs, files) **never talk directly to external parties**.

This architecture protects data and ensures that all transfers are **governed by clearly defined rules**.

Key Principles of the IDSA Approach

Let's break down a few of the most important design principles of this model:

1. Connector-Centric Design

Every organisation in the data space installs a **connector**—a lightweight software component that:

- Receives requests from outside parties,
- Applies the organisation's local policies,
- And then either shares the data, transforms it, or denies access.

This makes integration safe and consistent. Internal systems remain isolated, while the connector handles all communication with the outside world.

2. Usage Control, Not Just Access Control

Traditional systems focus on **access control**: either you have permission to see the data, or you don't.

IDSA goes further by introducing **usage control**:
Data is shared along with attached **obligations** and **restrictions** that must be followed even after the data has been delivered.

For example:

- "You may use this file, but you must delete it after 30 days."
- "You may not forward this data to third parties."
- "You may only use it for research purposes."

These rules are encoded as **machine-readable contracts** that travel with the data and are **enforced automatically** by compliant systems.

3. Trust Through Infrastructure

In open networks, trust doesn't just come from reputation—it must be built into the system.

That's why IDSA promotes a **Trust Framework**. This includes:

- **Identity Providers** – to verify who is making the request,
- **Certification Bodies** – to audit and attest that each connector behaves correctly,
- **Clearing Houses** (optional) – to record transactions and provide transparency or legal audit trails.

These entities work together to issue **cryptographic proofs**, allowing one connector to verify the identity and trustworthiness of another before sharing data.

A Mental Model: The Global Data Shipping System

Here's a simple metaphor to keep in mind:

> **An IDSA data space is like an international shipping system for data.**
> Every organisation has its own **customs office (connector)**.
> Data can't leave without:

- A **manifest** (usage policy),
- A **tamper-evident seal** (digital signature),
- And a **known destination** (a trusted peer connector).

This model ensures that data exchange is **intentional, auditable, and reversible**—a major leap forward from traditional API-based integration.

The Media Context Protocol (MCP)

If the **IDSA framework** explains *why* sovereign data spaces are important, then the **Media Context Protocol (MCP)** explains *how* this works in practice—especially when it comes to **media-rich data**.

MCP is a practical, lightweight protocol designed for the **structured exchange of media content**—including documents, images, audio, video, and even AI-generated summaries or metadata.

It defines a small number of clear, purposeful interactions that help external clients:

- Discover what files or resources are available,
- Request or subscribe to those resources,
- Ask the server to perform meaningful operations (like OCR or summarisation), and
- Understand what features the server supports.

This connector fully implements the MCP specification—and extends it in a few ways to support AI assistance, secure modes, and dynamic feature discovery.

How MCP Works: Building Blocks Explained

To understand MCP, it's helpful to look at its **four main building blocks**. Each one plays a specific role in making media exchange secure, structured, and flexible.

1. Resource Operations (resources/*)

What it does:
This set of calls allows a client to **list available files**, **read/download them**, or **subscribe to updates** if they change.

In our implementation:

- These operations are exposed through a `/mcp` JSON-RPC endpoint.
- The specific methods include:
    - `resources/list` – shows what's available.
    - `resources/read` – fetches the actual content.
    - **Server-Sent Events (SSE)** – enable live updates if something new appears or changes.

Think of this as the **"file explorer" interface** between systems.

2. Tool Operations (tools/*)

What it does:
Not all actions are simple downloads. Sometimes, a client wants the server to **do something meaningful** with the content—like extract text from a PDF, transcribe audio, or summarise a video.

These operations go beyond just transferring bytes—they involve **server-side logic**.

In our implementation:

- All available tools are stored in a **tool registry**, backed by a relational database (SQL).
- Each tool includes metadata describing:
    - What it does,
    - What input it expects,
    - What output it produces.
- This metadata is published using **JSON Schema**, so it's machine-readable by LLMs or other smart clients.

You can think of tools as **“remote functions”**—or a smart toolkit attached to your connector.

3. Prompt Templates (prompts/*)

What it does:
When clients use **Language Models (LLMs)** for analysis or summarisation, they often need help framing the right questions. MCP supports this by allowing connectors to offer **predefined prompt templates**.

In our implementation:

- The connector ships with ready-made prompt examples such as:
    - “Summarise this file”
    - “Identify all people mentioned”
    - “Summaries this image”
- These prompts are **parameterised**: the connector automatically injects relevant snippets from the file so the client doesn’t have to manually craft the input.

This helps standardise how LLMs interact with files, reducing errors and improving quality.

4. Capability Negotiation

What it does:
Not all clients and servers support the exact same features. Capability negotiation allows them to **agree on a common version** of the protocol and see what each side supports.

In our implementation:

- When a client sends an `initialize` request, it declares which protocol version it prefers.
- The server responds with the **highest compatible version** and a list of **feature toggles**.
- These details are also echoed back in the **HTTP response headers**, so even non-RPC-aware clients can inspect them.

This makes the connector **self-describing and adaptable** to different environments or client capabilities.

Why Use MCP Instead of REST APIs?

MCP is built on **JSON-RPC 2.0** and runs over plain HTTPS. This keeps it:

- **Simple to test** (you can use tools like `curl`),
- **Structured for automation**, and
- **Flexible enough for AI-powered clients**.

Compared to traditional REST APIs, MCP offers some powerful advantages:

Feature	Benefit
Stateful Sessions (via `Mcp-Session-Id`)	The server can remember who you are, what you've subscribed to, and what tools are available—without repeating it every time.
Event Streams (SSE)	Clients can **subscribe to live updates**—for example, if a new file is uploaded—**without needing WebSockets** or constant polling.
Schema-Aware Tool Calls	AI assistants or smart clients can **explore available tools, understand input formats**, and trigger the right function automatically—no guesswork.

MCP turns the connector into a **living interface**—something more like an app store or automation engine than a static API. It allows both people and machines to **discover, interact with, and benefit from media content**—in ways that are secure, compliant, and extensible.

FTI Mode Enhancements (Flanders Technology & Innovation)

While the Media Context Protocol (MCP) is inherently secure through transport-layer encryption (TLS), certain sensitive scenarios—such as the transfer of confidential R&D files, embargoed government material, or internal P&L documents—demand an even more hardened, zero-trust data flow.

To support these cases, the connector offers an **optional FTI mode**, designed in accordance with Flanders Technology & Innovation (FTI) standards. When enabled, this mode layers envelope-level cryptography, access controls, and fine-grained session enforcement over the standard MCP flow—without breaking compatibility with compliant clients.

What FTI Mode Adds: A Side-by-Side Comparison

Capability	Standard MCP	FTI Mode Additions
Transport	TLS 1.3	Payload-level AES-GCM encryption on top of TLS
File Transfer	download_file (base64 or raw)	secure_transfer tool with hybrid encryption (AES+RSA)
Auth	API keys by role	Requires API key with fti role + session must opt in
Session Model	Simple header-based	Two-step initialization: initialize → notifications/initialized
Rate Limiting	Redis-based per key	FTI keys often exempt to ensure uninterrupted secure flow
Audit Logging	API calls logged with IP + timestamp	+ mcp_sessions.fti_mode = true, integrity via triggers (planned)
Available Tools	All tools	Only FTI-approved tools (e.g., secure_transfer) shown

FTI mode transforms the connector into a crypto-first courier: every file becomes an individually wrapped, tamper-proof, recipient-only payload.

How FTI Mode Works

FTI Mode hinges on three key pillars:

1. **Session-based capability negotiation**
2. **Cryptographically secure file wrapping**
3. **Remote pull by trusted consumer MCPs**

Session Negotiation in FTI Mode

Session setup follows a strict two-phase handshake:

✅ Step 1: Client Requests FTI Mode

```
POST /mcp
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "initialize",
  "params": {
    "protocolVersion": "2025-06-18",
    "fti": true
  }
}
```

The **fti: true** flag and a valid **fti-role** API key are required. This creates a **pre-initialized session** in the database, stored with initialized = false.

✅ Step 1.2: The Server Validates the Request

Once the server receives the request, it performs two key checks:

- Is fti: true declared in the request?
- Does the client's API key include the special FTI role?

If both are valid:

- A new session is created in the database with the status initialized = false.

✅ Step 1.3: Server Responds

The connector returns:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "Mcp-Session-Id": "abc123",
    "protocolVersion": "2025-06-18",
    "capabilities": { ... }
  }
```

```
}
```

With headers:

```
Mcp-Session-Id: abc123
Mcp-Protocol-Version: 2025-06-18
```

This **Mcp-Session-Id** is used in all further communication. It binds the session to the specific capability set—including FTI mode.

✅ Step 1.4: Client Sends notifications/initialized

Next, the client sends a follow-up message (without a payload) to indicate that it's ready to proceed:

```
{
  "jsonrpc": "2.0",
  "method": "notifications/initialized"
}
```

The server marks the session as fully initialised by setting **initialized = true**. From this point onward, the session can use **secure_transfer**, access **FTI-only** tools, and skip throttling.

Why This Two-Step Dance?

You might wonder: why not just allow everything in one call?

This **"handshake"** pattern has important security and performance benefits:

- ✅ **Protects server resources:** If a client fails halfway through initialization, the session isn't marked as active, so no memory or quota is wasted.
- ✅ **Allows early rejection:** If the API key is invalid or not authorised for FTI, the server can block the session before any sensitive features are exposed.
- ✅ **Enables strict mode-switching:** Once a session is marked as FTI, it can't silently downgrade to a less secure mode—everything that follows is encrypted and traceable.

In summary, FTI mode gives you a hardened, compliance-ready version of the connector, with encryption and access control features suitable for regulated, confidential, or mission-critical environments.

Secure File Lifecycle in FTI Mode

When operating in **FTI mode**, the connector guarantees that sensitive files are not just transmitted over secure channels—but are also **cryptographically sealed** in a way that makes them readable **only by the intended recipient**.

This process involves several coordinated steps:

1. Key generation and rotation
2. Secure transfer initiation
3. File encryption (server side)
4. Decryption (client side)
5. Logging and audit trails
6. Threat mitigation at each layer

Key Generation

Each party (provider and consumer) must generate an RSA keypair:

Keys can be generated in two ways:

- Admin CLI tool, or
- API call to:

```
POST /rsa-keys/generate-keys
{
  "sender_id": "provider1"
}
```

🔐 **What happens under the hood:**

- A 2048-bit RSA keypair is generated.
- The private key is immediately wrapped using AES-GCM, with a master key (MASTER_KEY) securely loaded from environment variables.
- This ensures the private key never touches disk in unencrypted form.
- The public key, along with metadata (timestamp, key ID), is stored in the rsa_keys table.
- Public key is retrievable via /rsa-keys/latest or embedded in tool responses.

Automatic Key Rotation

To limit long-term exposure, the system supports **automatic key rotation**:

A **daily or weekly cron job** runs:

security/rotate_rsa_keys.py

- This script:
  - Generates a **fresh RSA keypair**
  - **Soft-deletes** older keys (so they're ignored in future transfers)
  - Respects a configurable retention window (KEY_RETENTION_DAYS, default: 90 days)

This keeps cryptographic hygiene tight while maintaining compatibility for recent transfers.

Secure File Transfer

To initiate a secure transfer, the provider sends a JSON-RPC request to upload a document on its server:

File Upload (Provider Side)

```
{
  "jsonrpc": "2.0",
```

```
  "id": 1,
  "method": "tools/call",
  "params": {
    "name": "secure_transfer",
    "arguments": {
      "action": "upload",
      "filename": "alpha.pdf",
      "description": "R&D Phase 1 results"
    }
  }
}
```

When the provider initiates a secure file upload using the **secure_transfer** tool with **action: upload,** the following cryptographic operations and system behaviors take place:

1. **AES Key Generation**
   A fresh, random 256-bit AES key is generated using **AESGCM.generate_key()**. This key serves as the symmetric key for encrypting the file content.

2. **File Encryption (AES-GCM)**
   The file's raw bytes are encrypted using AES-GCM, which provides both confidentiality and integrity in one operation. A 96-bit nonce is generated for this encryption. AES-GCM internally produces a 16-byte authentication tag, which is appended to the ciphertext. The resulting payload ensures tamper detection—if the file is modified, decryption will fail.

3. **Digest Computation**
   A SHA-256 digest is calculated over the plaintext file to support optional integrity verification during decryption (particularly in non-FTI modes or backward-compatible consumers).

4. **RSA Wrapping**
   The freshly generated AES key is then encrypted using the **consumer's public RSA key** (retrieved from the latest entry in the rsa_keys table). This is done using OAEP padding with SHA-256, ensuring cryptographic security and forward secrecy.

5. **Metadata Assembly**
   A metadata object is created containing:

   - The base64-encoded nonce
   - The base64-encoded digest
   - The wrapped symmetric key (re-encrypted with a master key before storage)

    - Original filename and optional description

6. This metadata is saved alongside the encrypted payload.

7. **Persistence**

    - The ciphertext (including AES tag) is written to disk under a UUID-named file.

    - The metadata is written to a .**meta.json** file with the same UUID prefix.

    - A corresponding row is inserted into the **encrypted_files** database table, including digest, filename, nonce, wrapped key, and file path—enabling retrieval, indexing, and future re-wrapping.

At this point, the file is **fully encrypted**, stored, and ready to be accessed securely by authorized consumers.

File Discovery (Consumer Side)

The consumer MCP calls:

```
POST /remote-files
{
  "remote_host": "backend-provider",
  "remote_port": 8000
}
```

When a consumer wants to retrieve files from a provider, they invoke the /remote-files endpoint. Internally, the following steps occur:

1. **API Key Resolution**
   The consumer supplies a remote_host and optionally an API key. If not provided, the system falls back to an environment variable or uses the consumer's own key.

2. **Remote MCP Client Bootstrapping**
   An MCPClient is instantiated targeting the provider's hostname and port. This client emulates a full JSON-RPC flow against the remote MCP instance.

3. **Secure Tool Invocation**
   The MCP client calls the secure_transfer tool on the provider, passing action: list.

4. **Remote Tool Resolution**
   On the provider side:

    - The request is received at /mcp, routed to handle_rpc_call(), and dispatched to call_tool_rpc().

- The secure_transfer tool returns a list of all available encrypted files, pulled from the encrypted_files table, and returns metadata such as file_id, filename, description, and sha256.

5. **Response Propagation**
   The list is returned back to the consumer's frontend, along with a timestamp indicating when the file catalogue was fetched.

This mechanism allows consumers to dynamically discover which files are available for secure transfer from a remote provider—without requiring hardcoded knowledge of the content or manual metadata exchange.

File Pull

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "tools/call",
  "params": {
    "name": "secure_transfer",
    "arguments": {
      "action": "download",
      "file_id": "abc123...",
      "remote_host": "backend-provider",
      "remote_port": 8000
    }
  }
}
```

Once a file has been discovered, the consumer initiates a pull-based request using the secure_transfer tool with action: download. Here's what happens behind the scenes:

1. **Request Routed to Provider**
   The consumer MCP instance forwards the download request to the provider's backend using an embedded call via MCPClient.

2. **File Retrieval and Key Wrapping (Provider Side)**
   On the provider:

   - The system looks up the file by file_id, either loading the ciphertext and metadata from disk or querying the encrypted_files DB.

- The previously wrapped symmetric key (originally stored encrypted with a master key) is decrypted using unwrap_key().

- This symmetric key is re-wrapped on demand using the consumer's public RSA key, provided inline with the request or fetched from the latest known key. This ensures that only the intended receiver can decrypt.

3. **Response Assembly**
   The provider returns a full payload including:

   - The newly wrapped AES key (specific to this consumer)

   - Nonce, ciphertext, and authentication tag (base64)

   - Filename and optional SHA-256 digest (optional but useful for validation)

4. **Transfer Completes**
   The consumer receives this payload and can now decrypt the file securely—without any direct disk access or SSH setup between the two backends. This is true zero-trust cross-server pull.

Decryption (Client Side)

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "method": "tools/call",
  "params": {
    "name": "secure_transfer",
    "arguments": {
      "action": "decrypt",
      "key_id": 2,
      "enc_sym_key_b64": "...",
      "nonce_b64": "...",
      "ciphertext_b64": "...",
      "sha256_b64": "..."
    }
  }
}
```

When the consumer wants to decrypt the received file, they send a request to **secure_transfer** with action: **decrypt**. Internally, this triggers the following flow:

1. **Key Lookup**
   The consumer supplies **key_id**, identifying which RSA private key to use. The system queries the rsa_keys table and **decrypts** the AES key using the RSA private key and OAEP padding.

2. **AES-GCM Decryption**
   Using the decrypted AES key and nonce, the ciphertext is decrypted with AES-GCM. If the ciphertext has been tampered with, the AES-GCM tag check will fail and raise an exception. This provides authenticated decryption out-of-the-box.

3. **SHA-256 Digest Verification (Optional)**
   If a sha256_b64 field was included in the request, the system re-computes the digest and checks for mismatches. This is an additional safeguard layer but not strictly needed when GCM integrity is present.

4. **Filename Resolution**
   The system attempts to reconstruct the original filename:

   - If provided: uses the client-supplied filename (sanitized).
   - Else: looks up by hash in the DB.
   - Else: generates a best-effort guess based on content type.

5. **Collision-Safe File Output**
   The file is saved to disk in the fti_decrypted folder. If the filename already exists, the system auto-increments the name to avoid overwriting.

6. **Final State**
   The file is now decrypted, verified, and safely stored locally—without ever exposing the AES key in transit or requiring the provider to retain decryption access.

Audit Logging and Compliance

All **tools/call** invocations are stored in **access_logs**:

Field	Description
api_key_id	Who called
endpoint	Always tools/call here
timestamp	Exact request time
ip_address	Client source

fti_mode	true if session used FTI

Admins can retrieve logs via:

```
GET /admin/trace
```

This returns all access events in JSON format, ready for SIEM ingestion or compliance review. **Only admin-key holders** may use this endpoint.

Threat Model & Mitigations

Threat	Standard Mode	FTI Mode Enhancements
TLS termination breach	TLS only	Encrypted payload + recipient-only RSA wrapping
Server compromise	File paths only	Wrapped AES keys; private keys are master-key encrypted
Session replay	Session tokens	Each payload is key-bound; cannot replay
File tampering	No native check	AES-GCM tag + optional SHA-256 digest validation
Role abuse	API key roles	fti-only access + filtered tool visibility

Summary

FTI mode doesn't just secure your API calls. It guarantees:

- Every file is **encrypted client-side**, using hybrid cryptography.
- Only the intended **RSA private key holder** can decrypt.
- Each exchange is **auditable, traceable**, and **bounded to its session**.

This makes the connector suitable for:

- R&D departments
- Government bodies
- Regulated industries
- Public-sector digital infrastructure

FTI mode = default security, turned up to eleven.

Even if an attacker compromsenrises the network, the database, or the server memory, they cannot decrypt past transfers—unless they also possess the partner's private key.

This approach delivers strong, forward-secrecy compliant protection suitable for regulated industries, R&D confidentiality, or public-sector secure collaboration.

Data Sovereignty & the Role of Connectors

One of the most important principles behind the FTI-compatible connector—and the broader data-space movement—is **data sovereignty**.

But what does that actually mean?

At its core, data sovereignty means that an organisation can **share access to insights without giving up ownership or control**. Just as a nation doesn't lose sovereignty when it lets people cross its borders with a visa, a company doesn't lose sovereignty when it lets someone temporarily access a document under agreed rules.

Data sovereignty ensures that:

- The organisation decides **who can access what**,
- Under **which conditions**,
- For **how long**, and
- With **proof** that these terms were respected.

This is not just a philosophical stance—it's baked directly into the technical design of the connector. Let's look at how that happens in practice.

How the Connector Enforces Data Sovereignty

The connector enforces sovereignty at three key levels: **technical**, **contractual**, and **accountability**.

1. Technical Control: Nothing Leaves Without Permission

By default, the connector restricts file access to a **controlled, whitelisted folder**

Only files within this directory (or its subfolders) are accessible via the connector.

To access a file, an external system must:

- Use a specially formatted path (e.g., file:///docs/paper.pdf)
- Pass through the connector, which enforces authentication and logging

Other types of access—like reaching directly into the file system—are blocked. Even someone with operating-system-level access can't retrieve the data unless they have the right permissions, **because access is filtered both at the OS level (ACLs)** and at the **protocol level (URI validation)**.

> 🔒 **No one gets in through the back door.** Every byte that leaves does so via the front gate—with a log and a lock.

2. Contractual Control: Not Just Access, But Purpose

Many systems today are designed around **access control**—you can either get the file or you can't.

Data spaces go further. They embrace the idea of **usage control**: the ability to specify **how the data may be used** even after it's received.

In future versions of this connector (aligned with IDSA roadmap), every resources/read response may carry an **attached usage policy**, describing terms like:

- "You must delete this file within 30 days"
- "You may not share this with third parties"

- "You may only use this for internal analysis"

These policies will be **machine-readable** and enforceable by downstream systems.

> ⚖️ Think of it like digital copyright terms that follow the file—and can be verified and enforced automatically.

3. Transparency & Accountability: Trust Built on Evidence

To truly enforce sovereignty, organisations must not only control and restrict access, but also **prove that policies were followed**.

That's why the connector provides:

- **Signed audit logs** – Every file access, tool call, or event is recorded with:
    - The requesting API key,
    - Timestamp,
    - IP address,
    - Action taken
- **Rate-limiting metrics** – Helps detect or prevent abuse by showing how often and how much a client is requesting
- **Notification hooks** – Optional real-time events (via `notifications/*`) can alert systems or administrators when a file is accessed, or a policy is triggered

This means the provider can show—to regulators, partners, or internal auditors—that:

- Only approved actions took place
- Only approved users accessed the data
- All access was **logged, rate-controlled, and optionally encrypted**

> **It's like a CCTV system for your data.** You don't just lock the door—you

prove it stayed locked.

Real-World Analogy: The Smart Parcel Locker

Think of a **smart parcel locker** at a train station.

- Only someone with the **correct one-time PIN (API key)** can open their assigned compartment.
- **CCTV (audit logs)** records when and how the locker was opened.
- The **parcel itself is vacuum-sealed** with a personalised code (FTI-mode encryption).
- If someone steals the parcel en route, it's worthless—because it can't be opened without the proper decryption key.

This is exactly how the FTI connector works.

- Files are protected physically (file path restrictions),
- Digitally (access controls),
- Contractually (future usage policies), and
- Cryptographically (FTI encryption)—
  all while maintaining transparency.

Why This Matters

In a world where data breaches, over-collection, and misuse are common, **data sovereignty restores balance**. It gives data providers the ability to share value—**without giving up power**.

The connector ensures that:

- **Data never leaks silently**, even during "approved" exchanges
- **Providers stay in control** at all times, with evidence to prove it

- **Compliance becomes proactive**—not reactive

For organisations operating in regulated sectors, collaborative ecosystems, or public-sector contexts, this isn't just desirable—it's essential.

System Overview

Now that we've explored how individual parts of the connector work, it's time to zoom out and look at the **system as a whole**.

This chapter covers:

- How the connector is structured at runtime
- How data flows through it, step-by-step
- What defences are in place to protect sensitive assets
- Where residual risks remain, and how we plan to mitigate them

This holistic view is critical for anyone deploying, securing, or integrating the system in production.

High-Level Architecture

At its core, the connector is a lightweight, self-contained web service. It's built to be simple to deploy, yet robust enough to support FTI-grade security guarantees.

Key Components

- **FastAPI Application**
  A single-process Python app serves all endpoints using FastAPI.
  By default, the app uses an in-process event bus, but it can be extended with Redis or NATS for distributed deployments.

- **PostgreSQL Database**
  Stores:

    - Identity (API keys and roles)
    - Cryptographic state (RSA key metadata)
    - Session lifecycle info (open MCP sessions)
    - Audit logs (access traces)
    - Encrypted files for Secure transfer
    - Tools that clients can use
    - Prompts that llm can use

- **Redis**
  Used only for stateless rate-limiting.
  If Redis becomes unavailable, the connector blocks traffic instead of failing **open.**

- **Flat-file Storage**
  Files are stored directly on disk for simplicity and transparency.
  Access is restricted via whitelisted paths and validated URIs.

- **Cloud AI Services (Optional)**
  The only outbound traffic is to cloud-hosted LLMs (e.g., Gemini).
  If no API key is provided, those tools are auto-disabled—ensuring no accidental leakage.

Default posture: least privilege, lowest trust, minimal dependencies.

Data Flow: Standard vs FTI Mode

Let's walk through the lifecycle of a request—from session creation to file delivery—under both **normal** and **FTI-secured** conditions.

Phase	Standard MCP Mode	FTI Mode (Enhanced)
A. Session	Client calls initialize, receives session ID, then sends notifications/initialized.	Same handshake, but server checks that the API key has the fti role. The session is flagged as FTI-bound and subject to restricted tool visibility.
B. Catalogue	Client lists tools via tools/list; sees general-purpose tools only (e.g., download_file, list_resources).	Client sees only FTI-approved tools like secure_transfer. Non-FTI tools are hidden unless explicitly allowed.

C. Request	Client calls tools/call with download_file, passing a URI. The file is returned in base64.	The flow splits between two parties: - Provider calls tools/call → secure_transfer with action: upload. - Consumer uses remote-files to discover, and later pulls via secure_transfer → action: download.
D. Processing	Server reads the file and encodes it in base64. No cryptographic sealing beyond transport-layer TLS.	On upload: • Server encrypts the file with AES-GCM under a random symmetric key. • That key is wrapped with the consumer's RSA public key. • Metadata (nonce, digest, etc.) is saved. • Row is persisted in encrypted_files. On download: • The AES key is rewrapped on-demand with the consumer's RSA key before transfer.
E. Response	Client receives JSON like: { filename, mimeType, data (base64) }	Client receives cryptographically sealed payload: { enc_sym_key_b64, nonce_b64, ciphertext_b64, sha256_b64, wrapped_sym_key_b64, filename }

F. Client Post-Process	Client decodes the base64 data to retrieve the raw file. No cryptography required.	Client uses its **private RSA key** to unwrap the AES key, decrypts the ciphertext using AES-GCM, and optionally verifies SHA-256. File is then written safely to disk.
G. Notifications	Optional: server emits change notifications via SSE stream (e.g. resources/updated).	Same system. SSE events are broadcast regardless of whether a tool is secure or standard. Event schema remains unchanged.

What's Actually Happening (FTI Mode Deep Dive)

- **Session**: FTI mode enforces a **two-phase init** (initialize + confirm) to lock in cryptographic expectations and avoid session downgrade attacks. Role mismatch or missing FTI intent immediately results in rejection.

- **Tool Visibility**: The MCP connector dynamically filters visible tools depending on session mode. In FTI sessions, only tools like secure_transfer and trusted metadata processors appear.

- **Upload (Provider)**:
  When secure_transfer → upload is called:

    - The provider **never stores** files in plaintext.

    - AES keys are generated per file.

    - Files are encrypted and saved as binary blobs on disk.

    - A SHA-256 digest is computed and logged.

    - AES keys are wrapped using the recipient's RSA public key and **re-wrapped with a master key for DB safety**.

- **Discovery & Pull (Consumer)**:

    - The consumer uses /remote-files to query the provider MCP's tool endpoint.

    - No direct disk or file system access is needed. Pulling files across firewalls or network zones becomes secure by design.

    - The consumer can **pull only what the provider explicitly exposed**, and receives a sealed file envelope.

- **Decryption**:
    - Consumer unwraps the AES key using its local RSA private key (retrieved from the DB by key_id).
    - AES-GCM ensures that any tampering (bit-flipping or substitution) will cause an authentication failure.
    - If provided, the SHA-256 digest allows for a secondary integrity check.
    - Files are written to a safe output directory, with auto-renaming to prevent overwrites.

Performance Overhead

Component	Estimate
AES-GCM (256-bit)	~1 GB/sec on modern CPUs
RSA-OAEP (2048-bit)	~3.5 ms per file
End-to-End Upload + Pull + Decrypt	Under 1s for files < 500MB

Despite the encryption, the latency overhead is minimal—ensuring FTI-mode is production-safe for high-volume systems.

Security Model & Threat Surface

Robust security isn't just about encryption—it's about clear **trust boundaries**, **attack surface understanding**, and **multi-layered mitigations**. The FTI Mode of the MCP connector is engineered to protect high-value digital assets from compromise even if individual layers are breached.

Trust Boundaries

Layer	Protection Measures
Public Internet	TLS 1.3 with HSTS; MCP endpoints reject plaintext HTTP requests
MCP Session	Session ID + version required in headers; FTI mode can't be downgraded once negotiated
Auth Layer	Identity is tied to API keys with role-based restrictions (fti, admin, standard)
App Logic	All tool invocations gated by session mode; no side effects unless session is active
Storage (Disk)	Files stored encrypted at rest using AES-GCM per file
Database	RSA private keys stored encrypted using a master key; no plaintext key material exists
Shell / Host OS	Master key (MASTER_KEY) is injected at runtime via environment variables; never persisted
Remote MCPs	No trust assumed—consumer MCPs cannot push, only pull; signatures verified dynamically

The connector assumes **zero trust** even in backend environments. Private keys, session tokens, and file artefacts are never left unprotected—even if an attacker compromises memory or disk.

Assets & Potential Adversaries

Asset	Why It Matters	Potential Threat Actor
Raw media files	May contain R&D, contracts, or confidential materials	External attackers, malicious insiders
AES keys	Grants access to decrypted file content	Memory snooping tool, server exploit
RSA private keys	Used to unwrap AES keys; compromise enables full file access	Compromised admin, leaked DB snapshot
API keys	Grants tool access and FTI mode entry	Phished user, Git repo leak, browser plugin
Audit logs	Prove compliance; allow backtracing of misuse	Rogue user seeking to cover actions
Session states	Bind capabilities and roles; needed to authorize requests	Attacker forging headers to hijack sessions

Mitigations Matrix

Threat	Control / Defense	Code Location / Strategy
Disk theft of private keys	RSA private keys are AES-wrapped using master key	security.encryption_utils.wrap_key()

Session spoofing or hijacking	Session ID + protocol version are required headers; server validates both	mcp_service.require_session()
API key abuse or brute-force	Rate-limiting enforced per key and tier using Redis	core.limiter.dynamic_rate_limit()
Unvalidated file reads	URIs normalized and base-path checked; no traversal allowed	core.uri_utils.normalize_uri()
Man-in-the-middle post-TLS	File content sealed using AES-GCM and RSA key-wrapping	services.file_crypto.store_encrypted_file()
Log tampering or manipulation	Logs stored in DB with FK to API key; roadmap: hash-chain / signed entries	services.trace_service.get_trace_logs()
Compromised MCP consumer	No file push allowed; provider encrypts-to-key only and never sees decrypted data	Pull-only model; no symmetric key exposure
LLM prompt injection	Inputs validated; no eval; tool access limited to llm role	tool_service._llm_process()

These controls are enforced **independently**. Even if one layer (e.g. TLS or Redis) is compromised, the file contents and session logic remain protected.

Residual Risks & Future Hardening

Residual Risk	Mitigation Strategy
LLM prompt-based exfiltration	Redact sensitive content before prompt injection, enforce strict schemas, or move LLM offline (on-prem)
Cloud LLM cost explosion (DoS vector)	Enforce tool quotas per API key; limit prompt size and token budget
Compromised MASTER_KEY (env var leak)	Emergency kill switch: revoke all API keys and RSA pairs, invalidate sessions, rotate master key at runtime
Lack of forward secrecy (RSA key reuse)	Implement rotating RSA keys per week; store encrypted historical keys with expiry metadata
Unverified file metadata	Use MAC or signature on metadata file (planned); prevent metadata forgery on rewrapped files

FTI mode is not just harder to break—it is designed to fail safely. Even if compromised, attackers must breach **multiple isolation layers** to reconstruct a single decrypted file.

Design Principles Recap

The connector is built on the following **security-by-design** principles:

- **Flat surface, deep defense**: Every boundary (transport, session, file, key) has its own independent crypto layer.

- **Zero-trust file exchange**: Files are encrypted for specific recipients using their public key. The server never retains decryption access.
- **No silent downgrades**: Once a session is FTI, it **cannot revert** to standard mode.
- **Composable tools, strict access**: FTI-only tools are never exposed unless the session is verified and marked.

In Summary

In Standard Mode	**Acts like a secure file-sharing assistant**
In FTI Mode	Becomes a digitally signed, RSA-wrapped, tamper-proof delivery vault. Only the recipient can open the envelope.

The system isn't just safe for engineers to build with—it's **auditable for regulators**, **trustable by governance teams**, and **future-ready for zero-trust infrastructure**.

DIAGRAM

Installation & Setup

This chapter explains how to go from a **blank machine** to a **fully running instance** of the FTI-Compatible MCP Connector. We'll walk through all the setup steps—**what's needed, why it's needed, and how to test that things work.**

> 🛠️ Most commands assume a Linux/macOS shell (bash, zsh). If you're using Windows, you can adapt these using WSL or PowerShell.

Prerequisites: What You Need Before You Begin

Before running the connector, you need a few core components installed. Here's a table with each dependency, the minimum version, and why it's important:

Component	Minimum Version	Why It's Needed
Python	3.11+	Runs the FastAPI server and all backend logic
PostgreSQL	14+	Stores API keys, RSA encryption keys, audit logs, sessions
Redis (optional)	7+	Tracks API rate limits (per-key, per-tier); blocks abuse
Build tools	`gcc`, `make`, `libffi-dev`	Used to compile cryptographic libraries (needed during `pip install`)
git	any	Used to clone the connector repository
curl or HTTPie	latest	Helps test the API locally with simple commands
OpenSSL CLI	1.1+	Used to generate test RSA keys during FTI-mode demo

Resource Requirements

- **Memory:** ~500 MB idle RAM usage
- **Disk:** ~50 MB for code and dependencies, plus space for the media files you manage (e.g., under `~/Desktop/FTI/MCP`)

Environment Configuration (.env)

The connector is configured using a .env file, which controls runtime behavior, cryptographic security, database access, and integration features.

This file **must** be customized per deployment: provider vs consumer, local vs production, or testing vs secure environments.

Setup Instructions

1. **Navigate to the repo root:**

```
cd fti_connector
```

2. **Copy the default env template:**

```
cp backend/utils.env .env
```

3. **Open .env in your editor and adjust settings as needed.**
   The defaults work for local dev, but **production setups must change key values.**

Key Environment Variables

Variable	Default / Example	What It Controls
ROLE	consumer or provider	Defines whether this node acts as a file **provider** or **consumer** in FTI mode
PORT	8000	HTTP port used by FastAPI (change if occupied)

DATABASE_URL	postgresql+psycopg2://fti_user:password@db-provider:5432/fti_provider	Connection string to Postgres DB (update host, password, DB name)
STORAGE_ROOT	/data/provider	Filesystem path where encrypted/decrypted files will be stored
MASTER_KEY	base64url-encoded 32-byte secret	Used to AES-wrap RSA private keys. Must remain secret and stable per deployment. Generate with: python -c 'import secrets, base64; print(base64.urlsafe_b64encode(secrets.token_bytes(32)).decode())'
API_KEY (optional)	bc2lepTJrzlJ2...	Hardcoded fallback API key for CLI testing or minimal auth environments

Advanced MCP & LLM Integration

Variable	Example / Default	What It Controls
MCP_URL	https://localhost:8000/mcp	MCP entrypoint
OLLAMA_ENDPOINT	http://localhost:11434/api/chat	Local LLM (Ollama) endpoint
OLLAMA_MODEL	hhao/qwen2.5-coder-tools:1.5b	Model identifier used for metadata extraction

GEMINI_API_KEY	AIza...	Enables Gemini-powered metadata enrichment tools
DEEPSEEK_API_KEY	***	Enables DeepSeek fallback for LLM-based tools
AUTO_METADATA	true	Auto-invokes metadata extraction when a resource is uploaded

Event Subscription & Realtime Features

These booleans toggle whether real-time updates (via SSE) are broadcast on tool or resource changes.

Flag	Effect
RES_SUBSCRIBE_ENABLED	Enables SSE push for resources/updated
RES_LIST_CHANGED_ENABLED	Enables broadcast when resources/list output changes
TOOLS_LIST_CHANGED_ENABLED	Enables tool catalogue update notifications
PROMPTS_LIST_CHANGED_ENABLED	Enables prompt catalogue update notifications

Key Retention & Rotation

Variable	Default	What It Does
KEY_RETENTION_DAYS	90	When rotating RSA keys, this value controls how long older keys remain valid for decryption.

⚠️ Notes & Tips

Port Conflicts: If 8000 is taken, override via:

```
export PORT=8080
```

- **Consumer & Provider Must Be Distinct**:
    - Use separate .env files per role (ROLE=consumer vs ROLE=provider)
    - Ensure DATABASE_URL and STORAGE_ROOT point to different backends or mounts.
- **Never check .env into version control** unless secrets are stripped.

Database & Redis Setup

PostgreSQL: Create the Database

```
sudo -u postgres psql <<'SQL'
CREATE USER fti_user WITH PASSWORD 'changeMe';
CREATE DATABASE fti_project OWNER fti_user;
GRANT ALL PRIVILEGES ON DATABASE fti_project TO fti_user;
```

This creates the credentials expected by your `.env` file.

Redis: Optional but Recommended

Redis tracks how often each API key is used. If it's missing, the rate limiter fails.

Ubuntu/Debian:

```
sudo apt install redis-server
sudo systemctl enable --now redis
```

macOS (Homebrew):

```
brew install redis
brew services start redis
```

If using cloud Redis (e.g. AWS ElastiCache), update **REDIS_URL** in **.env.**

Running the Service

Once everything is configured, run the connector:

```
# 1. Create a Python virtual environment
python3 -m venv .venv
source .venv/bin/activate

# 2. Install dependencies
pip install -r requirements.txt

# 3. Launch the FastAPI app
cd backend
uvicorn main:app --host 0.0.0.0 --port ${PORT:-8000} --workers 1
```

The connector is now running on http://localhost:8000.

You can test it by sending requests with:

- curl
- JetBrains REST client (*.http)
- Postman
- VS Code REST client extension

Quick-Start: Bootstrapping the System

This guide will take you from a clean setup to a working system for both:

- Standard MCP file usage
- FTI Mode: **provider** → **consumer** secure file exchange (via **secure_transfer**)

Create API Keys (One-time Setup)

```
# 1. Create the system-wide admin key
POST http://localhost:8000/admin/admin-key
```

This endpoint generates a single, privileged API key with administrator rights. You only need to call this once per deployment. The admin key is required for managing other API keys — including issuing, deactivating, or inspecting them. This key should be stored securely and never exposed in the frontend or to client users.

```
# 2. Create an FTI-mode key (admin only)
POST
http://localhost:8000/admin/keys/create?role=fti&rate_limit_tier=premium
&expires=2025-12-31
api-key: {{ADMIN_KEY}}
```

Creates an API key with the **fti** role, which grants access to advanced tools like **secure_transfer**. Only API keys with the admin role can create FTI keys. These keys are typically used in secure back-office systems, automated pipelines, or high-trust environments.

```
# 3. Create a standard key (admin only)
POST
http://localhost:8000/admin/keys/create?role=standard&rate_limit_tier=ba
sic&expires=2025-12-31
api-key: {{ADMIN_KEY}}
```

Issues a lower-privileged key suitable for day-to-day access to public tools such as **download_file, resources/list, and prompts/run.** These keys are rate-limited and meant for frontend UIs, cron jobs, or third-party integrations.

Generate Encryption Keys

```
# Generate RSA keypair (for decrypting encrypted files)
POST http://localhost:8000/rsa-keys/generate-keys
```

This generates a new RSA keypair (private + public key) and persists it in the system. The public key is used to encrypt files for your connector. Only the matching private key can decrypt them, ensuring confidentiality even if the file is intercepted. The returned key ID is used when performing a decryption call.

```
# Fetch the current public key again if needed
GET http://localhost:8000/rsa-leys/keys/latest
```

Fetches the latest RSA public key associated with your connector. This is useful when other services (like a provider) want to encrypt a file for you using the most recent available key.

Standard Mode: Base64 File Delivery

Use this when simple access is enough and encryption is handled by HTTPS.

A.1 – Start a Session

```
POST http://localhost:8000/mcp
Mcp-Protocol-Version: 2025-06-18
api-key: {{STANDARD_KEY}}

{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "initialize",
  "params": {
    "protocolVersion": "2025-06-18"
  }
}
```

Starts a new Media Context Protocol session using the standard flow. This session is not FTI-hardened but is protected by HTTPS and API key auth. You must include the returned **Mcp-Session-Id** in all future calls. It establishes a secure logical channel between client and server.

A.2 – Confirm Initialization

```
POST http://localhost:8000/mcp
Mcp-Session-Id: {{STD_SID}}
Mcp-Protocol-Version: 2025-06-18
api-key: {{STANDARD_KEY}}

{
  "jsonrpc": "2.0",
  "method": "notifications/initialized",
  "params": {}
}
```

Sends a notification to formally confirm the session initialization. This two-step handshake prevents half-open or zombie sessions. Until this is sent, most tool calls will be rejected. It's a required MCP convention for stability and traceability.

A.3 – Download File (Base64)

```
POST http://localhost:8000/mcp
Mcp-Session-Id: {{STD_SID}}
Mcp-Protocol-Version: 2025-06-18
api-key: {{STANDARD_KEY}}

{
  "jsonrpc": "2.0",
  "id": 3,
  "method": "tools/call",
  "params": {
    "name": "download_file",
    "arguments": {
      "uri": "file:///docs/alpha.pdf"
    }
  }
}
```

What this does:

This endpoint retrieves a file from the local filesystem and returns it to the client encoded in base64. It is used for fast, simple file access — especially when embedded transport (in JSON) is preferred over traditional streaming.

Internal Behavior:

1. **File Access**
   The tool safely resolves and opens the file at the given `uri` path. Unsafe paths or missing files raise an error.

2. **Base64 Encoding**
   The file's content is base64-encoded so it can be transported inside JSON over HTTP. This works well for small-to-medium files like PDFs, images, and logs.

3. **File Copy with Timestamp**
   In addition to returning the data inline, the system creates a timestamp-prefixed copy of the file and saves it to a persistent `downloads/` folder under your project root.
   This ensures:

   - Local traceability of what was accessed
   - Avoidance of filename collisions
   - Easier post-hoc inspection or auditing

4. **Structured Response**

   The returned payload includes:

```
{
  "filename": "alpha.pdf",
  "mimeType": "application/pdf",
  "size": 38472,
  "data": "JVBERi0xLjMKJcTl8uXr..."
}
```

⚠️ Security Note:

This method does **not encrypt** the file itself — it assumes a secure HTTPS channel is in place. For highly sensitive files, the `secure_transfer` tool in FTI mode should be used instead.

FTI Mode: Provider → Consumer Secure File Transfer

This is the **preferred mode for secure document delivery**. The provider encrypts the file for a known consumer. Only the consumer's private key can decrypt it.

B.1 – Consumer Starts an FTI Session

```
POST http://localhost:8001/mcp
Mcp-Protocol-Version: 2025-06-18
api-key: {{FTI_KEY}}

{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "initialize",
  "params": {
    "protocolVersion": "2025-06-18",
    "fti": true
  }
}
```

This request starts an FTI-mode MCP session, explicitly requesting a hardened protocol by setting **"fti": true.** The backend enforces that the API key has the **fti** role. FTI sessions enable restricted access to encrypted tool flows such as **secure_transfer**, enforce audit logging, and disable unapproved tool usage.

B.2 – Consumer Confirms Initialization

```
POST http://localhost:8001/mcp
Mcp-Session-Id: {{FTI_SID}}
Mcp-Protocol-Version: 2025-06-18
api-key: {{FTI_KEY}}

{
  "jsonrpc": "2.0",
  "method": "notifications/initialized",
  "params": {}
}
```

Same handshake as in Standard mode, but required for activating FTI tools. Until this step is acknowledged, even **secure_transfer** will be blocked. This ensures clarity and prevents accidental invocation of sensitive tools.

B.3 – Consumer Requests File from Provider

```
POST http://localhost:8001/mcp
Mcp-Session-Id: {{FTI_SID}}
Mcp-Protocol-Version: 2025-06-18
api-key: {{FTI_KEY}}

{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "tools/call",
  "params": {
    "name": "secure_transfer",
    "arguments": {
      "action": "download",
      "file_id": "abc123...",
      "remote_host": "backend-provider",
      "remote_port": 8000
    }
  }
}
```

This is the core of the secure exchange. The consumer (receiver) initiates a pull-based transfer from the provider. Here's what happens:

- The provider loads the requested file
- A random AES-GCM key is generated to encrypt the file contents
- This AES key is then encrypted with the consumer's RSA public key (provided in **pub_key**)
- A SHA-256 digest is computed to verify the ciphertext
- The full payload is digitally signed by the provider using its private key

The **consumer** receives:

```
{
  "encrypted_symmetric_key": "...",
  "ciphertext_b64": "...",
  "nonce_b64": "...",
  "tag_b64": "...",
  "digest_b64": "...",
  "provider_signature": "..."
}
```

The response includes:

- **encrypted_symmetric_key** (wrapped AES key)
- **ciphertext_b64** (encrypted file)
- **nonce_b64 and tag_b64** (GCM integrity/authentication)
- **digest_b64** (content integrity hash)
- **provider_signature** (digital signature for authenticity)

This guarantees that only the intended consumer can decrypt the file and verify its origin.

B.4 – Consumer Decrypts the File

```
POST http://localhost:8001/mcp
Mcp-Session-Id: {{FTI_SID}}
Mcp-Protocol-Version: 2025-06-18
api-key: {{FTI_KEY}}

{
  "jsonrpc": "2.0",
  "id": 3,
  "method": "tools/call",
  "params": {
    "name": "secure_transfer",
    "arguments": {
      "action": "decrypt",
      "key_id": 2,
      "enc_sym_key_b64": "...",
      "nonce_b64": "...",
      "ciphertext_b64": "...",
      "sha256_b64": "..."
    }
  }
}
```

The consumer uses this endpoint to decrypt the received encrypted package.
Internally:

- The AES key is unwrapped using the matching RSA private key
- The AES-GCM encrypted file is decrypted
- The SHA-256 digest is recomputed and checked

- The provider's digital signature is validated (to prevent tampering or impersonation)
- The file is stored to disk (e.g., under **/data/consumer/**) for access or audit

Real-Time Events

```
# Subscribe to file/resource changes
GET http://localhost:8001/mcp/stream
api-key: {{FTI_KEY}}
Mcp-Session-Id: {{FTI_SID}}
Mcp-Protocol-Version: 2025-06-18
Accept: text/event-stream
```

Subscribes to real-time server-sent events (SSE) for file changes, resource updates, and tool responses. This is useful for monitoring asynchronous activity such as when a file gets updated, or a tool was invoked. The stream pushes notifications over a long-lived connection, reducing the need for polling.

What You've Achieved

Step	Action	Result
1	Created Admin & User API Keys	Established secure roles: an admin key to manage the system, and scoped keys (standard, fti) to access tools.
2	Generated RSA Keypair	Created a cryptographic identity used to decrypt secure transfers; private key is stored encrypted server-side.
A.1–A.2	Initialized Standard MCP Session	Opened a standard protocol session and confirmed it, ensuring future tool calls are tied to a valid session context.
A.3	Downloaded File via download_file Tool	Accessed a local file, base64-encoded it, saved a timestamped copy in downloads/, and returned detailed metadata.

B.1–B.2	Initialized FTI (Secure) Session	Requested a secure session with fti=true, enabling access to sensitive tools like secure_transfer.
B.3	Pulled a File Securely from Provider	Fetched a file from another MCP instance; provider encrypted it using AES-GCM and the consumer's public key.
B.4	Decrypted the Secure File	Unwrapped the encrypted AES key, decrypted the file, verified authenticity and integrity, and restored the original file.
Bonus	Subscribed to Event Stream	Enabled real-time monitoring for file/resource changes via Server-Sent Events (SSE).

Technology Stack

This section catalogues the major components of the connector stack — backend frameworks, cryptographic choices, caching layers, and language model integrations — and explains why each was selected.

Where appropriate, we also highlight where these components can be swapped out to meet your organization's compliance, procurement, or scalability needs.

Backend Frameworks

These power the main HTTP interface and orchestration logic.

Layer	Component	Version	Purpose & Rationale

HTTP API	`FastAPI`	0.115.12	The main entrypoint. FastAPI provides automatic OpenAPI documentation, validation from type hints, and async support. Its concise routing syntax keeps endpoint files (like `router/mcp_router.py`) short and readable.
ASGI runtime	`Uvicorn`	0.34.x	Handles request concurrency and lifecycle. Works with asyncio and uvloop for low-latency responses. Production deployments can run multi-worker mode; `--reload` is used during local development.
Low-level base	`Starlette`	0.46.x	FastAPI is built on Starlette, but we also call it directly to implement features like streaming (Server-Sent Events) and custom TLS enforcement (`EnforceHTTPSMiddleware`).
ORM	`SQLAlchemy 2`	2.0.41	Used for all persistent data: API keys, session state, and RSA keys. Explicit transaction handling and typed models keep data logic testable and IDE-friendly.
CLI & Automation	`Typer`	0.12	CLI commands like `securitu/key_rotate.py` use Typer for consistent UX. Shares FastAPI's dependency-injection mindset, so config is centralized.

Flexibility tip: Any of these components can be swapped (e.g., Flask instead of FastAPI), but that would increase boilerplate. FastAPI's async+typed+docs trio is well-suited to secure API services.

Security & Cryptography Libraries

The MCP connector uses robust, real-world cryptographic techniques to secure sensitive data both **at rest** and **in transit**. Every file, key, and action is secured using well-audited standards, implemented through clean Python interfaces.

Concern	Library / Standard	Where Used	Details

Symmetric encryption	AES-GCM (256-bit) from cryptography	services/secure_file_service.encrypt_stream decrypt_stream	Each file is encrypted with a random 256-bit AES key and a 96-bit nonce. The output includes ciphertext, nonce, tag, and digest for authentication and integrity.
Key wrapping (at rest)	AES-GCM via wrap_key()	security/encryption_utils.py	Private RSA keys are wrapped with a MASTER_KEY before being persisted to the database. Key wrapping is authenticated and tamper-resistant.
Asymmetric encryption	RSA-2048 + OAEP (SHA-256)	secure_transfer_tool.py (during pull)	The AES key is encrypted using the recipient's public RSA key. This ensures only the holder of the matching private key can decrypt the file.
Digital signature	RSA-2048 + PKCS#1 v1.5 from cryptography	services/secure_file_service.sign_payload verify_signature	When a file is pulled securely, the provider signs the encrypted payload. The consumer verifies the signature using the provider's trusted public key from the rsa_keys table.
Digest (file fingerprint)	SHA-256 from hashlib	encrypt_stream()	A cryptographic digest of the plaintext is included to validate integrity after decryption. It serves as a second line of defense even if authenticated encryption fails.

TLS enforcement	EnforceHTTPSMiddleware + Reverse Proxy	main.py, .env	TLS 1.3 is enforced at the edge. Plain HTTP requests are rejected unless REQUIRE_HTTPS=false in local dev. HSTS headers are applied.
Randomness	secrets, os.urandom	All places where keys, nonces, or digests are created	All key material is derived from secure entropy sources — no weak PRNGs or deterministic values are used.
Rate limiting	Redis + custom rate tiers	core/limiter.py	Each API key is limited per tier (e.g. basic, premium). Helps prevent brute-force, LLM abuse, or key exhaustion attacks.

Why This Matters

Even if attackers gain access to the file system, database, or memory, they **cannot decrypt private keys or files** unless they also compromise the .env-held MASTER_KEY. All secrets are either ephemeral (per-request AES keys) or tightly bound to controlled keys.

Deployment Flexibility

If your organization uses a Key Management System (KMS) or Hardware Security Module (HSM) such as:

- **AWS KMS**
- **Google Cloud KMS**
- **HashiCorp Vault Transit**

…you can override wrap_key() and unwrap_key() in encryption_utils.py to delegate key protection externally. No architectural changes are required — just plug in your backend.

Messaging & Caching (Redis)

Redis is used narrowly and efficiently — no business logic depends on it, so outages degrade gracefully.

Feature	Implementation	Details
Rate limiting	`Redis 7 + core.limiter`	Tracks per-key quotas. Unlimited-tier keys (FTI) bypass the check. Burstable by adjusting TTLs.
Pub/Sub	*(planned)* Redis Streams or NATS	The current system uses an in-process event queue (`core.events`). Multi-replica deployments can switch to `XADD`/`XREAD`.
Caching	*(none yet)*	Media is streamed from disk on demand. LLM output caching may be added later using LRU in Redis.

Performance: Redis handles 10K+ rate-limit checks per second at sub-ms latency. Persistent state lives in PostgreSQL.

LLM Integration

The connector supports structured AI assistance, like auto-filling metadata based on PDF content. It is modular enough to work online or fully offline.

Aspect	Choice	Why
Primary API	`Gemini 1.5-flash (Google)`	Cheap, fast, supports JSON-structured replies out of the box — ideal for metadata extraction.

Fallback	`DeepSeek-chat`	Keeps things working when quotas are exhausted.
On-prem option	`Ollama + hhao/qwen2.5-coder-tools:1.5b`	Demonstrates offline function-calling. Used in `scripts/ollama_repl.py`.
Prompt interface	`services.llm_service.run()`	Manages fallback, retries, language detection, and format normalization.
Tool integration	`tool_service._llm_process`, `extract_metadata`	Hooks the LLM into MCP tool listing and dispatch.
Security rails	Truncation, schema validation, role checks	Prevents prompt injection and token runaway.

Resilience: If `GEMINI_API_KEY` is missing, LLM tools are auto-disabled — no crashes, no cost.

Summary Table

Stack Layer	**Technology**	**Can Replace With**	**Why Default is Chosen**
HTTP server	FastAPI + Uvicorn	Flask, Quart, or Django ASGI	Async + OpenAPI + typed + small footprint
Database ORM	SQLAlchemy 2	Tortoise, Django ORM	Explicit, typed, async-ready
Secure file crypto	AES-GCM + RSA-OAEP	Libsodium, Tink, AWS KMS	NIST-standard, portable, human-readable key formats

Key wrapping	AES-GCM w/ `.env` master key	Vault Transit, KMS	Simple demo setup; drop-in replacement possible
Rate limiting	Redis 7 (no persistence)	Memcached, in-process throttle	Fast and scalable; failure-safe
LLM support	Gemini / DeepSeek / Ollama	Azure OpenAI, Cohere, Claude	Gemini is fast & JSON-native; fallback ensures uptime

Core Features & API

This section unpacks the core functionality of the connector—how it handles sessions, enforces security, processes files, and allows AI-based interaction with user documents. If you've already followed the **Quick-Start**, you've seen some of these endpoints in action. Now we go behind the scenes to explain:

- **What happens when a session starts**
- **How API keys and roles control access**
- **How files are encrypted, downloaded, or decrypted**
- **How prompts are run using an internal AI engine**
- **How clients can subscribe to real-time file events**

The connector exposes a clean **Model Context Protocol (MCP)** API surface: structured, secure, and highly extensible. All interactions—whether downloading a file, running a prompt, or subscribing to file updates—use a consistent JSON-RPC format over HTTP. This lets clients build rich workflows without tightly coupling to backend logic.

The connector operates in two distinct modes:

Mode	Activation Method	Security Features	Example Use Cases
Standard Mode	Omit `"fti"` during `initialize`	TLS, API key auth, base64 downloads	Partner portals, quick previews, RAG
FTI Mode	Include `"fti": true` + FTI API key	Encrypted tools, digital signatures, RBAC	Cross-border transfer, research delivery

Each mode shares the same foundation (sessions, tools, resources) but enforces different rules depending on sensitivity and context. This modularity allows safe usage in public-facing environments while supporting deeply regulated workflows like secure scientific exchange.

In the following subsections, we'll explore the endpoints and mechanics of each capability. You'll see how seemingly simple API calls unfold into robust, verifiable, and secure operations under the hood.

ADMIN ENDPOINTS (prefix /admin)

Path	Method	Purpose / Action	Requires API Key	Role Required
/admin/key	POST	Generate or rotate **one** master administrator key.	No	–
/admin/health	GET	Verify caller has admin access and that the service is up.	Yes	admin
/admin/keys	POST	Create a new API key with a chosen role and rate tier.	Yes	admin
/admin/keys	GET	List every API key in the database (including inactive).	Yes	admin
/admin/keys/{key}	DELETE	De-activate (revoke) a specific API key.	Yes	admin
/admin/capabilities	GET	Return a JSON document describing server capabilities (tool list, limits, versions…).	Yes	admin

/admin/trace	GET	Download the security / access trace log.	Yes	admin
/admin/tools/register	POST	Register a new MCP tool **or** update an existing one.	Yes	admin

POST /admin/key

Creates the first or next admin key.
Typical one-time bootstrap call.

```
POST http://host:8000/admin/key
```

Response

```
{ "admin_key": "aaa1bbb2ccc3..." }
```

POST /admin/keys (create API key)

Query parameters

Name	Type	Default	Meaning
role	string	"standard"	Role for new key (must be one of database-backed roles).
created_by	string	–	Identifier of the human or service creating the key.
expires	string	–	Optional expiry date YYYY-MM-DD.
rate_limit_tier	string	"basic"	Attach a predefined rate limiter bucket.

Example

```
POST
"http://host:8000/admin/keys?role=admin&created_by=ops-bot&expires=2025-12-31"
  -H "api-key: {admin_key}"
```

POST /admin/tools/register

JSON body

```
{
  "name": "secure_transfer",
  "description": "Encrypted upload / download tool",
  "input_schema": { "action": { "type": "string", "enum":
["upload","download"] } },
  "binary": false,
  "fti_only": true
}
```

Returns the full stored tool record (id, timestamps, etc.).
If the tool already exists, the record is overwritten (upsert).

MCP CORE ENDPOINTS (root prefix /mcp)

Path	Method	Purpose	Requires API Key
/mcp	POST	Single JSON-RPC entry point for every MCP call.	Yes
/mcp	DELETE	Delete (terminate) the current MCP session. Needs Mcp-Session-Id header.	Yes
/mcp/stream	GET	Server-Sent Events channel delivering real-time notifications. Needs Mcp-Session-Id header.	Yes
/mcp/test-fire	POST	**DEV ONLY** – manually fire resource / tool change events.	No

POST /mcp (JSON-RPC)

Headers

```
api-key: {any_valid_key}
Content-Type: application/json
```

Example request : initialize session in FTI mode

```
{
  "jsonrpc": "2.0",
```

```
  "id": 1,
  "method": "initialize",
  "params": {
    "protocolVersion": "2025-06-18",
    "fti": true
  }
}
```

Successful response adds two headers

Header	Example value	Meaning
Mcp-Session-Id	38f2b882-c818-4…	Token required in subsequent calls.
Mcp-Protocol-Version	2025-06-18	Protocol version negotiated.

All subsequent JSON-RPC objects are sent to the same /mcp path and routed internally.

DELETE /mcp (end session)

Header **Mcp-Session-Id: {session}** must be present.
Returns HTTP 204 when the session is removed.

MCP EVENTS STREAM – GET /mcp/stream

Long-lived SSE connection streaming JSON payloads whenever:

- A resource list changes
- A tool list changes
- A prompt is added / modified
- A broadcast event originates from /mcp/test-fire

Headers

```
Mcp-Session-Id: {session}
Accept: text/event-stream
```

REMOTE FILE DISCOVERY (prefix /remote-files)

Path	Method	Purpose	Requires API Key
/remote-files/	POST	Query a **Provider** MCP to list encrypted files available for pull.	Yes

Request body fields

Field	Type	Meaning
remote_host	string	Hostname or IP of provider backend.
remote_port	int	Port (default 8000).
remote_api_key	string	Optional key for outbound call; if absent, env/API_KEY or caller's key is used.

Example

```
POST /remote-files
{
  "remote_host": "backend-provider",
  "remote_port": 8000,
  "remote_api_key": "provider-issued-api-key"
}
```

Typical Response

```
{
  "files": [
    { "file_id": "71a4...", "filename": "alpha.pdf", "description": "R&D
Phase 1", "sha256": "..." }
  ],
  "fetched_at": "2025-07-23T13:37:19Z"
}
```

RSA KEY MANAGEMENT (prefix /rsa-keys)

Path	Method	Purpose	Requires API Key
/rsa-keys/generate-keys	POST	Create a fresh RSA-2048 key pair for a sender identity.	No
/rsa-keys/keys/latest	GET	Return the newest public key (PEM) + id.	No

Generate Example

```
POST /rsa-keys/generate-keys
{
  "sender_id": "provider1"
}
```

Response

```
{
  "id": 12,
  "sender_id": "provider1",
  "public_key": "-----BEGIN PUBLIC KEY-----\nMIIBIjANBgkq...",
  "private_key": "-----BEGIN ENCRYPTED PRIVATE KEY-----\nMIIE..."
}
```

Latest-key Lookup

```
GET /rsa-keys/keys/latest
```

Returns

```
{
  "key_id": 12,
  "public_key_pem": "-----BEGIN PUBLIC KEY-----\nMIIBIjANBgkq...",
  "created_at": "2025-07-23T12:00:45Z"
}
```

DEVELOPER TEST EVENT – POST /mcp/test-fire

Body fields

Field	Type	Accepted values
kind	str	resources, tools, prompts, resource_updated
uri	str	Required when kind = resource_updated; ignored otherwise.

Purpose: trigger a fake broadcast so clients connected to /mcp/stream can test their SSE handlers.

Example

```
POST /mcp/test-fire
{
  "kind": "tools"
}
```

Response

```
 { "detail": "Triggered 'tools'" }
```

USAGE NOTES

1. Every endpoint that “Requires API Key” expects the header api-key: {value}.
2. Admin-only routes verify caller’s role via SimpleUser.role. Non-admin keys get HTTP 403.
3. For MCP RPC calls, always send or store Mcp-Session-Id returned from the initialize method; without it the server returns 400.
4. Tool registration (/admin/tools/register) is idempotent: same name will overwrite, new name will insert.
5. /remote-files is consumer-side only: providers never call it.
6. RSA private keys are returned on generation for bootstrapping; store them securely. They are never returned again.

Dual-Mode Operation

The connector is designed to operate in two distinct modes—**Standard Mode** and **FTI Mode**—each tailored for different levels of sensitivity and compliance. This dual-mode architecture allows organizations to use the same backend for casual, everyday interactions **and** for secure, audit-ready data exchanges without modifying the client code or infrastructure.

Mode Comparison

Mode	Activation Mechanism	Security Features Enabled	Intended Use Cases
Standard MCP	Omit `"fti"` in the `initialize` call	- TLS (HTTPS) - API key validation - Standard tools (e.g., download_file) - Logging	Partner portals, data previews, internal tools
FTI Mode	Include `"fti": true` and use FTI API key	- All Standard Mode features **plus**: - File-level encryption (AES-GCM) - Digital signatures - Tool restrictions based on role	Research delivery, regulated handoff, multi-org compliance

The key distinction lies in **what tools are enabled and how securely files are processed**. FTI mode introduces role-based access control, asymmetric encryption (RSA), and full lifecycle guarantees for secure file handoff and traceability.

What Happens Internally

When a client first connects, it must send an `initialize` request. This is the gateway to all other operations. Here's what occurs behind the scenes:

1. Session Creation

- The connector creates a new session record in the database.
- A unique session ID is issued, which must be included in all subsequent requests.

2. Mode Detection

- If the payload includes `"fti": true`, the connector switches into **FTI Mode**.
- Otherwise, it defaults to **Standard Mode**.

3. FTI Role Validation

- In FTI Mode, the API key must have the `"fti"` role.
- If the caller lacks this role, the session is rejected with a clear error.

4. Tool Filtering

- In **Standard Mode**, only general-purpose tools are enabled (e.g., `download_file`, `extract_metadata`, `resources/*`).
- In **FTI Mode**, tools like `secure_transfer` are made available **and** standard tools that don't meet security criteria are hidden.

5. Protocol Version Enforcement

- Clients must specify a supported MCP protocol version (e.g., `"2025-06-18"`).
- This ensures predictable compatibility and structured upgrades over time.

Real-World Examples

- **Standard Mode** is great for internal dashboards or quick retrievals: users can preview PDFs, fetch file summaries, or run AI prompts on images.
- **FTI Mode** is used when there's a compliance requirement to **prove who sent what**, **encrypt at rest**, and **ensure only authorized consumers can decrypt sensitive materials**.

Example `initialize` Request (FTI Mode)

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "initialize",
  "params": {
    "protocolVersion": "2025-06-18",
    "fti": true
  }
}
```

Headers Required:

```
POST /mcp
api-key: {{your-api-key-with-fti-role}}
Mcp-Protocol-Version: 2025-06-18
```

This request returns a `session_id`, which must be included in all future MCP calls. If the key lacks FTI role, or the `fti` flag is sent incorrectly, the session creation will fail with a clear explanation.

Authorization Framework

Every interaction with the MCP connector backend is **strictly protected by an API key**. These keys act like digital badges that verify the identity and privileges of the caller. Without a valid API key, no request will be processed — not even initialization.

This robust framework ensures **fine-grained access control**, **auditability**, and **defense against abuse or misuse**.

Key Attributes

Each API key is more than a simple token — it carries metadata and enforcement policies:

Attribute	Description

Active status	The key can be activated or deactivated without deletion.
Expiration	Keys can be configured to expire automatically after a certain period.
Roles	Controls what tools and modes the key can access: - `standard` – access to general tools - `fti` – access to secure tools - `admin` – full control over the system
Rate Limits	Keys can be assigned usage tiers to prevent overuse or abuse.

How Authorization Works

Each request to the MCP backend must include the `api-key` header. For example:

```
api-key: {{YOUR_KEY}}
```

Behind the scenes, every request goes through a multi-step validation:

✅ **Step 1: Is the key present?**

If not, the request is rejected immediately with a 401 Unauthorized error.

✅ **Step 2: Is the key active and not expired?**

The backend checks whether the key is still valid. Deactivated or expired keys are denied access.

✅ **Step 3: Is the caller allowed to perform this action?**

Each key's **role** determines what methods and tools are available:

- A `standard` key **cannot** use tools like `secure_transfer`.
- A `fti` key **can** access both secure and standard tools, depending on session mode.
- An `admin` key has full access to key management and trace logs.

✅ **Step 4: Log the request**

All validated requests are logged, including:

- IP address of the caller
- Time and date of the request
- Endpoint and tool accessed

This creates a clear, immutable audit trail for security reviews and usage analytics.

Example Authorization Headers

```
POST /mcp
api-key: sk_test_standard_abc123
Mcp-Session-Id: {{session-id}}
Mcp-Protocol-Version: 2025-06-18
```

Admin Controls

The connector includes dedicated endpoints for administrative key management. These are only accessible with API keys assigned the `"admin"` role:

Endpoint	Description
`POST /admin/keys/create`	Creates a new API key with specific role and tier

`POST /admin/keys/deactivate`	Deactivates a specific API key (without deleting it)
`GET /trace`	Returns full audit logs of API usage (per key, endpoint, timestamp, etc.)

These controls allow system administrators to:

- Safely rotate keys
- Revoke compromised credentials
- Monitor usage patterns and detect anomalies

Why This Matters

A flexible authorization system ensures:

- Only **approved clients** can access sensitive functionality.
- Access can be tuned per user, role, and tool.
- Every operation is traceable, helping meet compliance requirements (e.g., auditability under ISO 27001 or GDPR).

This framework provides the foundation for both **secure multi-tenant environments** and **fine-grained operational trust**.

Secure File Transfer (FTI Mode Only)

This feature enables **secure, encrypted file sharing between two systems** (a *provider* and a *consumer*), typically across backend boundaries. The process ensures **only the intended recipient can decrypt** the data. It uses **hybrid cryptography**, combining:

- **RSA (asymmetric)** encryption for key exchange and digital signatures
- **AES-GCM (symmetric)** encryption for efficient file encryption

Who are the actors?

- **Provider**: The system that owns or sends the file (e.g., an R&D lab).
- **Consumer**: The system that wants to securely receive and decrypt the file (e.g., a regulator or partner).

Step-by-Step Protocol

1. Generate RSA Key Pairs (Both Sides)

Each side needs a **long-term RSA key pair**. These are stored securely in the database and used for encryption, decryption, and digital signature verification.

Provider:

```
POST /rsa-keys/generate-keys
{
  "sender_id": "provider1"
}
```

Consumer:

```
POST /rsa-keys/generate-keys
{
  "sender_id": "consumer1"
}
```

Technical Detail:
Keys are RSA-2048, and private keys are encrypted at rest using a `MASTER_KEY` from environment config.

2. Consumer Initializes MCP Session in FTI Mode

The consumer initiates the session, requesting **FTI mode**, which enables secure tools and enforces role restrictions.

```
POST /mcp
api-key: {{api-key-provider}}
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "initialize",
  "params": {
    "protocolVersion": "2025-06-18",
    "fti": true
  }
}
```

Explanation:
This tells the backend: “I’m starting a secure session and I’m allowed to use FTI tools.”
The backend will now restrict visible tools to only those marked `fti_only: true`.

3. Backend Sends Initialized Notification

This step completes the handshake. It informs the backend that initialization is done.

```
POST /mcp
api-key: {{api-key-provider}}
Mcp-Session-Id: {{mcp-session-id-pro}}
{
  "jsonrpc": "2.0",
  "method": "notifications/initialized",
  "params": {}
}
```

4. Provider Uploads a File (Encrypted at Rest)

The provider uploads a file using the `secure_transfer` tool with `action: upload`.

```
POST /mcp
api-key: {{api-key-provider}}
Mcp-Session-Id: {{mcp-session-id-pro}}

{
```

```
  "jsonrpc": "2.0",
  "id": 1,
  "method": "tools/call",
  "params": {
    "name": "secure_transfer",
    "arguments": {
      "action": "upload",
      "filename": "Meeting_1.pdf",
      "description": "Daily trading P&L report - 17 Jul 2025"
    }
  }
}
```

Under the Hood:

- File is encrypted using a **random AES-256-GCM key**
- That AES key is **encrypted (wrapped)** using the consumer's RSA **public key**
- A **SHA-256 digest** of the file is computed for integrity
- The entire payload is **digitally signed** using the provider's RSA **private key**
- All of this is stored on disk (e.g., `fti_encrypted/`)

5. Consumer Lists Available Remote Files

To discover files, the consumer queries the provider over a public endpoint:

```
POST /remote-files
api-key: {{api-key-consumer}}
{
  "remote_host": "backend-provider",
  "remote_port": 8000
}
```

Response includes metadata like `file_id`, filename, timestamp, and description.

6. Consumer Downloads the File

Now the consumer initiates a secure pull using `action: download`.

```
POST /mcp
api-key: {{api-key-consumer}}
Mcp-Session-Id: {{mcp-session-id-con}}

{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "tools/call",
  "params": {
    "name": "secure_transfer",
    "arguments": {
      "action": "download",
      "file_id": "2edcc454ea4741f2b51ad96f2e5237cc",
      "remote_host": "backend-provider",
      "remote_port": 8000
    }
  }
}
```

Returned:

```
{
  "ciphertext_b64": "...",
  "enc_sym_key_b64": "...",
  "nonce_b64": "...",
  "tag_b64": "...",
  "sha256_b64": "...",
  "provider_signature": "...",
  "key_id": 2
}
```

7. Consumer Decrypts the File

Now the consumer decrypts the response using their **RSA private key** and verifies authenticity:

```
POST /mcp
api-key: {{api-key-consumer}}
Mcp-Session-Id: {{mcp-session-id-con}}
```

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "method": "tools/call",
  "params": {
    "name": "secure_transfer",
    "arguments": {
      "action": "decrypt",
      "key_id": 2,
      "enc_sym_key_b64": "...",
      "nonce_b64": "...",
      "ciphertext_b64": "...",
      "sha256_b64": "JsnzP42YJVd7clQtNHliOQiyBrw3Hx77FkqvJ3vEmtM="
    }
  }
}
```

What happens:

- Decrypt RSA-wrapped AES key
- Decrypt file with AES-GCM using nonce and tag
- Verify SHA-256 hash matches original digest
- Optionally, validate provider's digital signature

Why This Is Secure

Layer	Technology	Purpose
AES-256-GCM	Symmetric encryption	High-speed, secure file encryption
RSA-2048 + OAEP	Key wrapping	Ensures only intended recipient can read the AES key

SHA-256	Integrity check	Detects tampering or corruption
RSA Signatures	Authenticity	Verifies file came from provider and wasn't altered
API keys	Access control	Only valid users can interact with tools

Notes for DevOps & Architects

- File metadata and encrypted payloads are persisted on disk and/or in the DB.
- Keypairs are stored securely with encryption-at-rest using a master key.
- All file actions are logged with IP, session ID, and timestamps.
- You can rotate keys or use external HSMs (e.g., Vault) by extending `wrap_key()` and `verify_signature()`.

Resource Catalogue & Subscriptions

The connector exposes a **real-time, queryable catalogue of files** available to a client within their current session. This enables powerful functionality such as:

- Browsing available reports
- Reading structured file contents
- Reacting to changes in shared folders
- Keeping UI dashboards in sync

These actions are **fully secure**, governed by the current session and the user's API key role.

What is a "Resource"?

A **resource** refers to any file the system knows about — for example:

- A PDF uploaded by the provider
- A text transcript generated by AI
- An encrypted file awaiting download
- A summary result or prompt output

The connector maintains a **normalized URI** for each resource, like:

```
file:///docs/Meeting_1.pdf
file:///summaries/report.json
```

All access to resources is read-only and must pass **security filtering**, ensuring users only see files they're authorized for.

Supported Methods

Method	Purpose
`resources/list`	Lists all files accessible to the current user
`resources/read`	Reads the content of a given file (text or JSON)
`resources/subscribe`	Subscribes to file updates (via server events)
`resources/unsubscribe`	Cancels a previous subscription

Example: List Available Files

```
POST /mcp
api-key: {{api-key}}
Mcp-Session-Id: {{session-id}}

{
  "jsonrpc": "2.0",
  "id": 10,
  "method": "resources/list",
  "params": {}
}
```

Server Response:

```
{
  "resources": [
    {
      "uri": "file:///docs/Meeting_1.pdf",
      "mimeType": "application/pdf",
      "size": 32768,
      "lastModified": "2025-07-21T12:34:56Z"
    },
    {
      "uri": "file:///summaries/Meeting_1.json",
      "mimeType": "application/json",
      "size": 1567,
      "lastModified": "2025-07-21T12:35:10Z"
    }
  ]
}
```

Explanation:

- Every listed file has a normalized URI.
- MIME type helps UIs decide how to display or preview.
- Modification date supports caching or invalidation.

Example: Read a File's Contents

```
POST /mcp
api-key: {{api-key}}
Mcp-Session-Id: {{session-id}}

{
  "jsonrpc": "2.0",
  "id": 11,
  "method": "resources/read",
  "params": {
    "uri": "file:///summaries/Meeting_1.json"
  }
}
```

Response:

```
{
  "content": {
    "summary": "The meeting covered Q2 losses and next steps..."
  },
  "mimeType": "application/json",
  "uri": "file:///summaries/Meeting_1.json"
}
```

Notes:

- This call is optimized for **text- and JSON-based files**.
- For binary downloads (e.g., images or encrypted blobs), use the `download_file` or `secure_transfer` tools.

Subscribe to Resource Updates

If your app needs to **react in real time** (e.g., new file available or re-uploaded), you can subscribe to a resource.

```
POST /mcp
api-key: {{api-key}}
Mcp-Session-Id: {{session-id}}

{
```

```
  "jsonrpc": "2.0",
  "id": 12,
  "method": "resources/subscribe",
  "params": {
    "uri": "file:///docs/Meeting_1.pdf"
  }
}
```

Event Notification Format

When the file changes (e.g., re-encrypted, replaced), the server will send:

```
{
  "jsonrpc": "2.0",
  "method": "notifications/resources/updated",
  "params": {
    "uri": "file:///docs/Meeting_1.pdf"
  }
}
```

Real-World Use Case:
Imagine a regulator frontend that subscribes to reports. As soon as the provider re-uploads or re-encrypts a file, the UI updates without refreshing.

Unsubscribe from Updates

To stop receiving events:

```
POST /mcp
api-key: {{api-key}}
Mcp-Session-Id: {{session-id}}

{
  "jsonrpc": "2.0",
  "id": 13,
  "method": "resources/unsubscribe",
  "params": {
    "uri": "file:///docs/Meeting_1.pdf"
  }
}
```

Access Filtering and Safety

Internally:

- All `uri` values are **sanitized and normalized** using strict file system rules.
- Access is filtered using **session context and API key role**.
- Malicious inputs like `file:///../../etc/passwd` are **automatically rejected**.

Developer Tips

- You can subscribe to entire folders (e.g., `file:///docs/`) in future releases.
- Combine `resources/list` + `subscribe` to build live dashboards.
- JSON files returned by `read` can be parsed or piped into LLM-based prompts.

Tool Registry & Execution

At the heart of the connector lies a powerful and flexible **Tool Registry**. Tools are discrete, named units of logic — essentially, callable modules that perform a task. Some tools fetch files, others encrypt them, and others analyze content using LLMs.

Think of tools like "microservices inside the connector" — each one does one thing, well, securely, and in a role-aware fashion.

What is a Tool?

A **tool** is any callable action the connector exposes over MCP. These include:

Tool Name	Description	Availability
`download_file`	Retrieves a local file and returns it as base64	All modes

`secure_transfer`	Encrypts, signs, and transfers files	FTI mode only
`extract_metadata`	Analyzes file content (text/image)	All modes
`llm_process`	Invokes an LLM with a prompt	All modes

Some tools are **restricted to FTI Mode** because they involve sensitive operations (e.g., secure file exchange). Others are more general-purpose and always available.

Listing Available Tools

To explore which tools your current session can access:

```
POST /mcp
api-key: {{your-key}}
Mcp-Session-Id: {{session-id}}

{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "tools/list",
  "params": {}
}
```

Example Response:

```
{
  "tools": [
    {
      "name": "download_file",
      "description": "Download a file"
    },
    {
      "name": "secure_transfer",
      "description": "Encrypted file exchange",
```

```
      "fti_only": true
    }
  ]
}
```

Note: You'll only see tools your session is allowed to access.

If you're in standard mode, FTI-only tools will be **excluded from the list entirely**.

Calling a Tool

To invoke a tool, use the `tools/call` method and specify:

- the tool name
- its expected arguments

For example, to download a file:

```
{
  "jsonrpc": "2.0",
  "id": 4,
  "method": "tools/call",
  "params": {
    "name": "download_file",
    "arguments": {
      "uri": "file:///Meeting_1.pdf"
    }
  }
}
```

Typical Tool Response:

Each tool defines its own output. For example, `download_file` returns:

```
{
  "filename": "Meeting_1.pdf",
  "mimeType": "application/pdf",
```

```
  "size": 12890,
  "data": "JVBERi0xLjQKJc... (base64)",
  "saved_path": "/downloads/20250721_Meeting_1.pdf"
}
```

Other tools like `secure_transfer` or `llm_process` may return:

- encrypted artefacts
- LLM-generated summaries
- metadata dictionaries

Behind the Scenes: Tool Execution Flow

Every time a tool is called, the connector follows this **secure execution pipeline**:

1. **Tool Lookup**
    - Confirms the tool exists and is enabled in this mode.
2. **Role & Session Check**
    - Validates that your session has permission to access this tool.
    - Example: FTI tools are blocked unless you initialized with `fti: true` and have an `fti` role API key.
3. **Schema Validation**
    - Each tool defines its expected arguments as a JSON schema.
    - If you forget a required field (like `uri`), the call fails with a clear error.
4. **Dispatch**
    - Internally, the connector routes your call to the correct implementation.
    - This could be:
        - A local file reader (`download_file`)

        - A secure encryption service (`secure_transfer`)
        - An embedded LLM (`llm_process`)
        - A prompt execution engine (`prompts/run`)
5. **Logging**
    - Every call is logged, including:
        - API key used
        - IP address
        - Tool name and parameters (scrubbed if sensitive)
        - Timestamp
    - This helps with auditability and debugging.

Tool Modularity: How It's Built

Each tool is defined as a Python class implementing the `ToolInterface`, e.g.:

```
class DownloadFileTool(ToolInterface):
    name = "download_file"
    input_schema = { "uri": {"type": "string"} }
    ...
```

On application boot, all tool classes are **discovered automatically** and registered.

You can extend the connector with new tools (e.g., `anonymize_file`, `translate_document`) by simply adding a class to the `handlers/` directory.

Security Notes

- FTI-only tools **cannot be invoked** unless the session is in FTI mode and the API key has the `"fti"` role.

- The `tools/list` method respects this constraint by hiding tools the caller isn't allowed to see.
- Each tool can choose whether to:
    - Allow or reject certain file types
    - Write audit entries
    - Generate server-side copies of results (e.g., ZIP files or reports)

Real-World Examples

Scenario	Tool	Notes
Preview a report	`download_file`	Returns base64 + metadata
Pull a secure contract	`secure_transfer`	Requires encrypted download + decrypt
Summarize a document	`llm_process`	Routes to a configured LLM and returns a structured reply
Extract metadata from an image	`extract_metadata`	Uses DeepSeek/Gemini LLM with fallback
Name participants from meeting PDF	`prompts/run`	Invokes a prompt template with analysis logic

Prompts & LLM Utilities

Prompts are reusable natural language templates that can extract or summarize information from text or image files using a local LLM. They allow users to run sophisticated analysis on

files with a single method call—whether it's naming meeting participants from a PDF or describing what's inside an image.

Features

- Prompt definitions are synced from code into the database at boot.
- All prompts are listed via prompts/list.
- Clients can preview a prompt's structure with prompts/get.
- A full LLM-based analysis is performed via prompts/run.

Built-in Prompts

Name	Description	Input Type	Returns
summarize-file	Summarizes the content of a document (PDF/text).	text/pdf	JSON { summary }
name-participants	Lists participant names mentioned in the document.	text/pdf	JSON ["Alice", "Bob"]
summarize-image	Describes an image in plain language.	image (JPEG/PNG)	JSON { summary }

Request/Response Examples

List all available prompts

```
POST /mcp
api-key: {{api-key-fti}}
Mcp-Session-Id: {{mcp-session-id}}
Mcp-Protocol-Version: {{mcp-protocol-version}}

{
  "jsonrpc": "2.0",
```

```
  "id": 20,
  "method": "prompts/list",
  "params": {}
}
```

Response:

```
{
  "prompts": [
    {
      "name": "summarize-file",
      "description": "Summarise the content of a given file...",
      "arguments": [{"name": "uri", "required": true}],
      ...
    },
    ...
  ]
}
```

Get full prompt details (for UI preview)

```
POST /mcp
api-key: {{api-key-fti}}
Mcp-Session-Id: {{mcp-session-id}}
Mcp-Protocol-Version: {{mcp-protocol-version}}

{
  "jsonrpc": "2.0",
  "id": 21,
  "method": "prompts/get",
  "params": {
    "name": "name-participants",
    "arguments": {
      "uri": "file:///Meeting_1.pdf"
    }
  }
}
```

Returns the text template, argument schema, and example response format (e.g., a JSON array of names). Also includes an inlined representation of the target file (text or base64 image).

Run the prompt (LLM invocation)

```
POST /mcp
api-key: {{api-key-fti}}
Mcp-Session-Id: {{mcp-session-id}}
Mcp-Protocol-Version: {{mcp-protocol-version}}

{
  "jsonrpc": "2.0",
  "id": 22,
  "method": "prompts/run",
  "params": {
    "name": "name-participants",
    "arguments": {
      "uri": "file:///Meeting_1.pdf"
    }
  }
}
```

Returns:

```
{
  "participants": ["Alice", "Bob", "Charlie"]
}
```

Under the Hood

- Text-based prompts are formatted and fed directly to the LLM with an appended response_hint.
- Image prompts use base64 encoding and are passed as visual context using multi-message roles.
- Fenced or malformed JSON output is cleaned and parsed before being returned.
- Any parse error or unrecognized content type is returned with an error field.

Extending Prompts

To add new prompts:

1. Define the prompt in core/prompt_defs.py.

2. Include template, return_key, response_hint, and required arguments.
3. On app boot, all prompt definitions are auto-synced into the DB.
4. The prompt can now be listed, previewed, and invoked via MCP.

Example prompt definition:

```
{
  "name": "detect-languages",
  "description": "Detect all natural languages used in a document.",
  "arguments": [{"name": "uri", "required": True}],
  "template": "Which languages are used in this document?",
  "return_key": "languages",
  "response_hint": "Return a JSON array of language names."
}
```

Rate Limiting & Audit Logging

To ensure the connector remains fast, secure, and resistant to misuse, it employs a **two-pronged safeguard**:

1. **Rate Limiting** – to prevent denial-of-service (DoS) or accidental overuse
2. **Audit Logging** – to maintain a verifiable history of all access and activity

Together, these features help uphold performance and traceability in both development and production environments.

Rate Limiting

Rate limiting ensures that clients can't overwhelm the server by sending too many requests too quickly — whether due to a bug, brute-force attack, or misconfigured integration.

How It Works

- The connector uses a **Redis-backed fixed-window algorithm**.

- Each API key has its own independent request counter.
- When a client sends a request, the system checks:
    - How many requests have already been made within the current window (e.g., last 10 seconds)
    - Whether the client has exceeded its quota

Technical Details

Feature	Description
Algorithm	**Fixed-window counter** (e.g., max 100 requests per 10 seconds)
Unit of limit	Enforced **per API key**, not per IP address — avoids false limits in NAT/proxy setups
Response on breach	Returns `HTTP 429 Too Many Requests` with a JSON error body
Exemptions	API keys with `tier = "unlimited"` (e.g., admin or trusted FTI roles) bypass rate limits
Persistence	Rate counters are stored in **Redis**, ensuring accuracy even across multiple servers

Example Error Response on Limit Exceeded:

```
{
  "detail": "Rate limit exceeded. Please wait before retrying."
}
```

This helps client developers build retry-aware systems and prevents excessive load during normal use.

Why Per-API Key?

Many systems rate-limit based on **IP address**, but this creates major issues:

- **Corporate networks**: Dozens of users may share one external IP
- **Mobile devices**: IPs can rotate quickly
- **VPNs / proxies**: Mask real origin

By using **API keys as the identifier**, the connector ties usage limits to the **authenticated identity**, not the network path — resulting in both better security and more predictable user experience.

Audit Logging

Every interaction with the connector — successful or not — is logged in a persistent, queryable audit log. This log captures:

Field	Meaning
`endpoint`	The API endpoint (e.g., `/tools/call`, `/admin/keys/create`)
`timestamp`	UTC time the request was received
`ip_address`	IP of the requester

`api_key`	The API key used for the request

Even requests that are **rejected or fail validation** are still logged.

Example Entry (from `/trace`):

```
{
  "endpoint": "/keys/create",
  "timestamp": "2025-06-30T08:55:31.520Z",
  "ip_address": "127.0.0.1",
  "api_key": "F4SxyRREnTvXav6iFp_31ZoYjleWrcQGa6dnr7zw-_Y"
}
```

These logs are critical for:

- **Audit trails** in regulated environments
- **Security investigations** if an API key is compromised
- **Debugging** permission or tool access issues

Logs can be queried by admin users using:

```
GET /trace
api-key: {{admin-key}}
```

Admin Controls and Observability

Administrators can inspect, limit, or disable keys using dedicated endpoints:

Method	Purpose
`POST /admin/keys/create`	Create API keys with defined roles/tiers

`POST /admin/keys/deactivate`	Revoke keys immediately
`GET /trace`	View all recent request logs

This allows for **tight governance** over how the connector is accessed — including revoking a misused key in real time.

Summary – Core Features & API

This section outlined the internal capabilities that power the connector's secure and modular architecture. Whether operating in a low-security preview mode or a tightly regulated FTI environment, the connector offers fine-grained control, rich automation tools, and strong observability.

Capability	What It Enables	Why It Matters
Dual-Mode Operation	Clean separation between Standard and FTI workflows	Avoids accidental use of secure tools in general-purpose flows and vice versa
Authorization Framework	Role-based access control using signed API keys	Enforces tool visibility, rate limits, and privileges with cryptographic safety
Secure File Transfer	End-to-end encrypted delivery using AES-GCM and RSA	Ensures only the intended receiver can access confidential payloads — even if servers are compromised
Resource Catalogue & Subscriptions	Real-time access to local files and events	Enables clients to react automatically when files are added, modified, or deleted

Tool Registry & Execution	Dynamic discovery and use of callable tools	Makes the platform extensible, automatable, and easy to integrate into pipelines or assistants
Prompts & LLM Utilities	Natural-language powered file analysis with pre-defined templates	Allows users to extract, summarize, or interpret files (PDFs, images, text) with zero ML knowledge
Rate Limiting & Audit Logging	Protection against abuse and full traceability	Prevents misuse while maintaining clear records of who accessed what, when, and how

Together, these features make the connector secure by default, but flexible enough to power advanced tooling, real-time workflows, and compliant data exchange in sensitive environments.

Implementation Details

Understanding how the connector works — file-by-file, service-by-service.

This section offers a guided tour of the internal structure of the MCP connector backend. You can think of this like opening the hood of a well-architected machine and labeling the key components — the ignition, the gears, the dashboard wiring — while also learning how they interact to deliver secure and reliable behavior.

The codebase is structured into clearly separated directories, each with its own responsibility. This modular design not only promotes maintainability and scalability, but also makes it easy to onboard new developers or auditors.

Directory Structure

The backend folder follows a **clean and layered architecture**, where every major concern (e.g., APIs, security, tools, prompts) has its own well-labeled “department.” Here is an overview of the main directories and what they do:

Path	Purpose
`router/`	Public-facing HTTP routes (e.g., `/mcp`, `/rsa-keys`) – defines the **entry points** to the system.
`handlers/`	Contains **tool handlers** – each tool (e.g., `download_file`) has a dedicated implementation.
`services/`	The **core business logic** layer. All meaningful operations (e.g., encryption, sessions, LLM use) live here.
`core/`	Glue logic and internal registries – manages tools, prompts, events, config, and global capabilities.
`db/`	Defines database models (tables), repository logic, and DB connections. Everything related to persistence.
`security/`	All cryptographic and HTTPS logic: RSA, AES-GCM, key signing, encryption, HTTPS enforcement.
`interfaces/`	Contains **abstract base classes** and typed interfaces that define consistent internal APIs.
`schemas/`	Typed request/response schemas used in tool arguments, validation, and JSON-RPC processing.
`flow/`	Scripts for custom run-flows, integration tests, REPLs, or demos (e.g., connecting to a Google Drive MCP).

`services/remote_mcp_client.py`	Connects to other MCP-compatible servers to **use their tools**, like file listing/search in a remote system (e.g., Google Drive). Not used for encrypted file pull.
`http_tester/`	HTTP client scripts for manual testing (e.g., using `.http` files in IntelliJ/PyCharm).
`tests/`	Unit tests and integration tests for all critical paths, including FTI flows and tool registration.
`shared_data/`	Internal encrypted and decrypted files – e.g., files moved during secure file transfer.
`main.py`	The **entry point** of the backend server. Starts the FastAPI app.
`.env*`	Environment variable configurations for different environments (provider, consumer, development).

Mental Model

Think of each directory as a department in a company:

- **router/** is reception (public entry),
- **handlers/** are specialized workers (tools),
- **services/** is operations (core logic),
- **core/** is central management (registries, glue),
- **security/** is IT & Compliance (crypto),
- **db/** is Finance (data records and logs),
- **main.py** is the CEO pressing the power button.

Key Modules & What They Do

Each module below is part of the internal machine. Together, they allow the connector to support FTI-grade secure file exchange, standard metadata tools, audit logging, prompts, and more.

Area	Module(s)	Purpose and Explanation
API Routing	`router/*.py`, `handlers/*.py`	These define the **public HTTP interfaces** for every feature — including session initialization, tool calls, RSA key exchange, admin routes, etc.
Session Handling	`services/session_service.py`	Creates and tracks **MCP sessions**, validates protocol versions, manages role bindings and FTI toggles.
Tool Management	`services/tool_service.py`, `core/tool_loader.py`, `core/tool_registry.py`, `services/tool_registry_service.py`	Loads available tools, validates input, and enforces tool-level permissions.
Prompt Execution	`core/prompt_defs.py`, `prompt_loader.py`, `services/prompt_service.py`, `db/prompt_repo.py`	Defines and runs **LLM prompts**, syncing natural-language templates into the database and allowing rich queries over files.
File Access	`services/resource_service.py`, `shared_data/`	Reads and writes files from local storage, and tracks

		decrypted/encrypted states in FTI flows.
Encryption & Signing	`security/encryption_utils.py`, `file_crypto.py`, `signing.py`	Handles hybrid AES-GCM + RSA encryption, digital signature creation and validation. Secures files for FTI delivery.
Key Management	`router/rsa_keys.py`, `services/key_service.py`, `security/rotate_rsa_keys.py`, `db/rsa_key_repo.py`	Allows generation, rotation, storage, and secure wrapping of RSA key pairs.
Remote Connections	`services/remote_mcp_client.py`	Connects to **other MCP servers** to **use their exposed tools** (e.g., `file_search`, `read_file`). It does not perform secure file pulls.
Rate Limiting	`services/redis_rate_limiter.py`, `core/limiter.py`	Uses Redis to enforce per-user rate limits. Designed to prevent spam or API abuse.
Audit Logging	`services/trace_service.py`, `db/trace_repo.py`	Records all API activity — including failed calls, rate limit violations, or internal errors.
Authorization	`core/auth.py`, `security/security.py`	Handles API key lookup, role enforcement (`standard`, `admin`, `fti`), and authentication checks.

Event Broadcasting	`core/events.py`	Sends **real-time file updates** (e.g., a document was modified) using Server-Sent Events (SSE).
Schemas & Interfaces	`schemas/mcp.py`, `interfaces/*.py`	Ensure consistency of input/output types, tool interface definitions, and abstract layers for service injection.
App Launch	`main.py`	Starts the FastAPI app, loads routes, config, tools, LLMs, keys, and SSE streams.

Database Schema

The connector uses **PostgreSQL** to store all persistent data. Every secure interaction — from verifying API keys to tracking encrypted file transfers — is backed by a structured and auditable database layer.

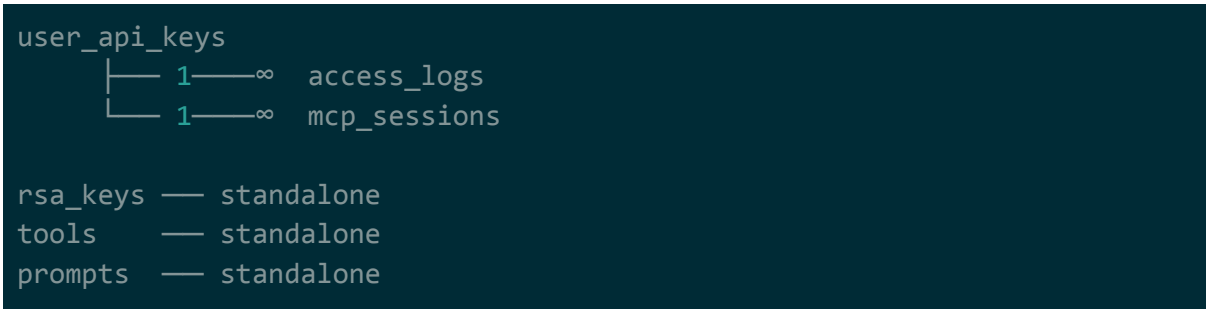
This schema is intentionally **minimal and purpose-driven**, allowing for fast lookups, easy traceability, and safe deletion or rotation of sensitive information.

Core Tables & What They Store

Table	Key Columns	Purpose
`user_api_keys`	`id`, `key`, `role`, `expires_at`, `tier`, `active`	Stores all issued API keys with their security role (`standard`, `admin`, or

		`fti`), expiration, rate tier, and status.
`access_logs`	`id`, `api_key_id`, `endpoint`, `timestamp`, `ip_address`	Every single API call is logged here for **auditing**, whether it succeeds or fails.
`mcp_sessions`	`id`, `session_id`, `role`, `fti_mode`, `initialized`, `created_at`	Tracks active sessions (standard or FTI), so that tool access and capabilities are session-bound.
`rsa_keys`	`id`, `sender_id`, `public_key_pem`, `encrypted_private_key`, `created_at`, `soft_deleted`	Stores public + encrypted private RSA key pairs used in **secure file transfers**, signed payloads, and remote verification.
`tools`	`id`, `name`, `description`, `fti_only`, `input_schema`, `output_schema`	Defines all **registered tools** and their availability: e.g., some tools like `secure_transfer` are marked `fti_only`.
`prompts`	`id`, `name`, `description`, `template`, `args_schema`, `auto_run`, `created_at`	Stores LLM prompt templates and parameter schemas. Used to dynamically generate file summaries, captions, or search queries.

Relationships Diagram (Simplified)



- **One API key** may be responsible for **many access logs** and **many sessions**.
- **RSA keys**, **tools**, and **prompts** are independent tables — they are loaded by services, not tightly coupled to users or sessions.

Automatic Table Creation on Startup

The connector uses **SQLAlchemy's ORM layer** to define the database schema. All tables listed above are declared as Python classes, inheriting from `Base` (the declarative base). These include `UserAPIKey`, `AccessLog`, `McpSession`, `RSAKey`, `Tool`, and `Prompt`.

When the FastAPI application starts, it automatically **creates all required tables** in the connected PostgreSQL database if they do not already exist.

This is done via the following line in `main.py`:

```
Base.metadata.create_all(bind=engine)
```

This command performs the following:

- **Scans all SQLAlchemy models** registered with the `Base` declarative base.
- **Checks the target database** (via `engine`) to see which tables already exist.
- **Creates any missing tables** using the appropriate PostgreSQL DDL (`CREATE TABLE`).
- **Ensures compatibility** between the declared schema and the actual structure in the database (non-destructively).

This approach ensures that:

- **First-time deployment is seamless**: the database is initialized automatically.
- **No raw SQL scripts are needed** during installation.
- **The schema is always in sync** with the application code — unless models are later refactored without migration tooling.

> Note: For production environments with evolving schemas, it's recommended to adopt **Alembic** for managed migrations. But for internal tools or plug-and-play deployment, `Base.metadata.create_all()` offers a robust zero-touch solution.

Why This Matters (For Non-Technical Readers)

- **Auditability**: Every action — file access, prompt run, tool call — is stored in a **tamper-resistant** log.
- **Traceability**: You can trace who accessed what, when, and from where — critical for FTI-grade compliance.
- **Security Isolation**: Even if an attacker steals a database snapshot, they cannot decrypt RSA keys without the `.env`-held master key.
- **Session Safety**: Each client session is tracked independently. Once a session ends, its FTI privileges or tool access expires.

Security & Key Management

Security is not an afterthought in this connector — it's a **core design principle**. Every sensitive operation, from file upload to tool access, is protected using strong cryptography, proper key isolation, and strict role enforcement.

This section explains how cryptographic protection works, how keys are managed, and what keeps your encrypted data safe even if the system is compromised.

1. Master Key: The Root of All Trust

The `.env` file contains a special variable:

```
MASTER_KEY="..."
```

- This key is **never exposed** to clients or stored in the database.
- It is used to **wrap (encrypt)** all RSA private keys before they are written to disk or stored in the database (`rsa_keys` table).

If this key is missing or invalid:

- The connector **refuses to start** — this avoids running in an insecure state.
- If stolen, encrypted keys stored in the DB would be **useless** without the master key.

2. RSA Key Generation & Rotation

To support secure file transfers and digital signatures, the system uses **RSA public/private key pairs**.

Each backend (e.g. provider, consumer) generates its own RSA key via:

```
POST /rsa-keys/generate-keys
{
  "sender_id": "provider1"
}
```

- The public key is stored in plaintext.
- The private key is encrypted using the master key and stored as `encrypted_private_key`.

You can rotate keys using the script:

```
python backend/security/rotate_rsa_keys.py
```

This:

- Creates a fresh RSA key pair.
- Soft-deletes keys older than 90 days.
- Keeps encrypted private keys safe for decryption & signing.

3. Secure File Encryption: Hybrid Model (AES + RSA)

FTI-mode file transfers use **hybrid encryption** — combining:

- **AES-GCM (symmetric)** for speed and large payloads.
- **RSA (asymmetric)** to encrypt the AES key.

How it works:

When a file is uploaded using `secure_transfer`:

1. **AES Setup**
    - A random 256-bit AES key and nonce are generated.
    - The file is encrypted with AES-GCM → produces:
        - `ciphertext_b64` (encrypted data)
        - `tag_b64` (integrity check)
2. **RSA Wrapping**
    - The AES key is encrypted using the **consumer's RSA public key**.
    - The encrypted AES key is stored as `enc_sym_key_b64`.
3. **Digital Signature**
    - The full payload is **signed** by the provider's private RSA key.
    - This proves authenticity and integrity on the receiver's side.
4. **Decryption**
    - The consumer verifies the signature and SHA256 digest.
    - Then decrypts the AES key using their **private key**.
    - Finally, decrypts the file using AES-GCM.

This ensures:

- Only the **intended receiver** can decrypt the file.
- No **man-in-the-middle** can tamper with or forge payloads.
- Even internal staff **without the private key** cannot decrypt files.

4. API Key Validation: Enforcing Access Control

All incoming requests must include an `api-key` in the header. Example:

```
api-key: bc2lepTJrzlJ2m7r2oOPgrDkd28WBxY1MgWmfLSNCxs
```

The validation process checks:

- Does this key exist in `user_api_keys`?
- Is it active and not expired?
- Is it allowed to access the requested tool/method?

If **any of these fail**, the request is blocked with:

- HTTP `403 Forbidden`
- JSON-RPC error: `-32602 Invalid params`

⛔ FTI tools like `secure_transfer` are **completely hidden** from non-FTI sessions, even if requested manually.

5. HTTPS-Only Mode (Production Safe Defaults)

By default, the server **refuses to run** in production over plain HTTP.

Configuration:

```
REQUIRE_HTTPS=true
```

If a request is received without TLS:

- The connector **rejects the request**.
- It returns an error saying HTTPS is required.

It also sets strict headers:

```
Strict-Transport-Security: max-age=63072000; includeSubDomains; preload
```

This protects against downgrade attacks and insecure proxy misconfigurations.

Summary

Feature	How it Works
Master Key	Wraps all private RSA keys with a local secret; system refuses to boot without it
RSA Key Rotation	Secure 2048-bit key pairs, stored encrypted; rotates every 90 days via script
AES+RSA Hybrid Crypto	Encrypts large files efficiently, protects secrets using public key crypto
Signature & Integrity	Every encrypted payload is digitally signed and hashed
Strict HTTPS Mode	TLS required for all connections in production; adds HSTS headers
API Key Enforcement	Keys are role-validated and expire; used to control who sees what

Error Handling & Protocol Rules

Software can fail — that's expected. What matters is *how* it fails.

The connector is designed to **fail predictably, transparently, and safely**, with errors that are:

- Meaningful for developers (through consistent codes)
- Secure (no sensitive info exposed)
- Traceable (recorded in logs)

This section explains how errors are handled across transport (HTTP) and protocol (JSON-RPC) layers.

Error Design Principles

Principle	What It Means
Consistent JSON structure	All responses, even errors, follow a standard JSON-RPC format.
No leaks	Errors never reveal private data (e.g. file paths, stack traces).
200 OK ≠ success	A JSON-RPC error still returns HTTP 200 — this is part of the protocol spec.
Transport errors are real	Issues like rate limits or missing headers return proper HTTP codes (e.g. 429).

Error Format (JSON-RPC)

Every JSON-RPC error looks like this:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "error": {
    "code": -32602,
    "message": "Invalid parameters: missing 'uri'"
  }
}
```

Field	Meaning
`code`	A numeric code (see table below)
`message`	Human-readable explanation
`id`	Matches the original request (or `null`)

Common Error Scenarios

Scenario	HTTP Code	JSON-RPC Code	Thrown From
Missing or invalid API key	403	-32602	`guard_mcp()` (auth guard)
Unknown RPC method	200	-32601	`api.mcp_router`
Malformed or missing arguments	200	-32602	`tool_service.validate_arguments`
File not found	200	-32601	`resource_service.read_file()`
Rate limit exceeded	429	-32602	`core.limiter`

Internal server error	200	-32001	Catch-all (FastAPI exception)
Invalid session	403	-32602	`session_service.validate()`

Note: These are always **logged** in `access_logs`, including failed ones.

Why HTTP 200 for Errors?

According to the JSON-RPC 2.0 specification, all application-level errors (e.g. bad arguments, missing tools) must still return `HTTP 200 OK`, unless the entire request is malformed at the transport level.

This means:

- A valid JSON-RPC call with the wrong tool → 200 + error object
- A missing `api-key` header → HTTP 403 + plain JSON error
- A rate-limited client → HTTP 429 + plain JSON error

Typed Exceptions Internally

Internally, the backend raises:

- `HTTPException` for transport-level issues (like 403, 429)
- `JSONRPCException` (custom) for protocol-level problems

Example:

```
if not session:
    raise JSONRPCException(-32602, "Invalid session or expired")
```

This keeps layers separate:

- Security logic never touches UI
- Protocol logic never leaks system internals

Summary Table

Feature	Value
Protocol	JSON-RPC 2.0 (over HTTP)
Error design	Transparent, safe, and structured
Validation layers	Key, session, role, tool, schema
Logging	All errors recorded with timestamp, IP, key
Spec-compliant	200 for JSON-RPC errors, 4xx for transport failures

Example: Rate Limit Breach

Request:

```
POST /mcp
api-key: too-fast-key
```

Response:

```
HTTP/1.1 429 Too Many Requests
{
  "detail": "Rate limit exceeded: 100 calls per 10 seconds"
}
```

Logged as:

```
{
  "timestamp": "2025-07-22T12:03:18.021Z",
```

```
  "api_key": "too-fast-key",
  "endpoint": "/mcp",
  "ip_address": "192.168.0.12"
}
```

Implementation Philosophy: Decoupled, Modular, Replaceable

This section goes beyond *what* the connector does — it explains *how* it's built underneath. If you've ever maintained a large codebase, you know: technical debt creeps in fast. This connector avoids that trap by being built around core principles that promote long-term **maintainability, security, and portability**.

Let's unpack what that actually means — for developers, DevOps, and even non-technical stakeholders.

1. Everything is Modular

The connector is not a monolith. It's a **collection of focused, interchangeable modules**, each with a single purpose.

Module	Role
`services/`	Encapsulates logic — from session management to file encryption
`api/` / `router/`	Handle HTTP and JSON-RPC requests — entry point for clients
`security/`	Manages cryptographic operations, key rotation, signature verification
`core/`	Contains glue code: event broadcasting, tool registry, rate limiting

`db/`	Defines persistent models and repositories using SQLAlchemy

This allows you to swap or extend components without touching others:

- Replace **FastAPI** with **Flask**? Easy — just rewrite `main.py` and the `api/` folder.
- Switch from **PostgreSQL** to **MySQL**? Update `db/database.py`.
- Want to **inject a custom LLM backend**? Extend `llm_service.py`.

> *Result: You can evolve each subsystem independently. You're not locked into any tech stack or vendor.*

2. Minimal Global State

Many bugs stem from global variables or shared state that bleeds across requests.

Here, **every request is isolated**:

- No globals except for `SessionLocal()` database connections.
- All logic depends only on the incoming request, headers, and parameters.
- Event systems like `core.events` use pub-sub rather than shared memory.

> *Why it matters: Two users accessing the system at the same time will never interfere with each other. It's safe for concurrency, threads, and async.*

3. Secure by Default

The connector assumes a hostile environment — even internally.

Every entry point is validated:

- API keys must be present, active, and non-expired.
- Each method checks roles (`admin`, `standard`, `fti`).
- Argument schemas are validated on every tool call.

- File paths are resolved safely to prevent traversal attacks.

Encryption is **always on in FTI mode**:

- Files are encrypted with AES-256-GCM and wrapped with RSA before any remote transfer.
- Keys are rotated and stored encrypted in the database.

> *"Security by default" doesn't just mean HTTPS — it's embedded into tool design, storage logic, and request validation.*

4. Strict Validation at Boundaries

All external input is treated as untrusted — even from internal tools.

- JSON schemas are defined per tool (e.g., `download_file`, `secure_transfer`).
- Errors are raised early — before DB calls or file reads happen.
- Malformed calls return safe JSON-RPC errors, not stack traces.

Example:

```
if not path.is_file():
    raise HTTPException(404, f"File not found: {path}")
```

Bad inputs are like fire — they're contained before they spread.

5. Decoupled Interfaces

Internal components use **interfaces, not concrete classes**. This is seen in:

Interface	Used By
`ToolInterface`	All tools (`download_file`, `secure_transfer`, etc.)

`RateLimiterInterface`	Swappable Redis-based limiters
`PromptInterface`	Any prompt loader or runtime (JSON, file-based, etc.)

This design:

- Allows you to plug in new backends without rewriting logic
- Enables clean mocking during tests
- Makes behavior predictable across deployments

6. Replaceable, but Sensible Defaults

The system runs **out of the box**, but doesn't limit you.

You get:

- Redis-based rate limiter — but you can replace it with your own token bucket
- JSON file prompt loader — but can switch to DB-based or remote templates
- Local file access — or plug into a cloud storage provider using a remote MCP client

You're in control — but never starting from scratch.

Summary Table: Design Pillars

Principle	Description
Modular structure	Each function has its own folder, test, and interface

Isolation by design	Every request runs clean, no shared memory mess
Secure by default	Validation, encryption, key rotation, HTTPS enforced
Swappable components	Replace DB, crypto, or LLM layers without rewriting everything
Fail early, fail safely	Consistent error handling, clean exceptions
Built for humans	Easy to navigate, audit, and extend without fear

End-to-End Scenarios

Real systems aren't understood by reading API specs in isolation. They're understood by following the life of a request — how it starts, what the system does, and what the user sees.

This section illustrates how the connector works in practice, using three realistic scenarios that mirror common organizational workflows. Each example ties together:

- The **client-facing action**
- The **internal steps across modules**
- The **database or logging side-effects**
- The **security, performance, and usability implications**

Rather than repeat prior technical descriptions, we connect them into **narrative flows**, with links to implementation details where appropriate.

Scenario A — Downloading a File in Standard Mode

> *Use case: A dashboard system needs to let users download a compliance report without encryption — just access control.*

The Story:

A client establishes a basic MCP session (`fti: false`) and sends a `tools/call` to download a PDF. This could be a public annual report or a daily sales extract for partners.

What Happens:

1. The connector checks the API key and session.
2. It verifies the requested file exists and is whitelisted.
3. It reads the file from disk, encodes it in base64, and returns it over HTTPS.
4. A log entry is created in `access_logs` with timestamp, API key, and file.

What the Client Sees:

A downloadable PDF. If they're using Postman, it's decoded automatically. If they're scripting it, they decode the base64 and save it.

> 🔍 See: Section 8.5 for tool dispatching, Section 9.4 for download logic.

Scenario B — Secure Transfer in FTI Mode

> *Use case: A regulated institution shares a financial contract with a specific recipient. It must be encrypted for that recipient alone.*

The Story:

The provider uploads a file using `secure_transfer` with a standard `upload` call. Later, a designated consumer authenticates, lists files available from that provider, and pulls it securely via `download` — triggering encryption with their public key.

What Happens:

1. Both parties generate RSA keypairs and store them securely.
2. The consumer initiates an FTI session using `initialize` with `fti: true`.
3. The provider uploads the file.

4. When the consumer pulls it, the file is encrypted on the fly with AES-GCM, and the symmetric key is wrapped with the consumer's RSA key.

5. The encrypted payload is delivered. The consumer uses their private key to decrypt it.

What the Client Sees:

A secure bundle that only they can decrypt. Even if intercepted, the file is unreadable. The entire exchange is logged for auditing.

Scenario C — AI-Powered Metadata Extraction

> *Use case: Before archiving a folder of meeting transcripts, a user wants to auto-generate summaries and action points.*

The Story:

A user uploads a document (e.g., a meeting PDF), and then calls `extract_metadata`. The connector reads the content, selects the right prompt based on file type and language, sends it to a lightweight LLM (e.g., Gemini 1.5 or DeepSeek), and returns structured metadata.

What Happens:

1. Text is extracted (via PyMuPDF or plain read).

2. The language is detected.

3. A context-aware prompt template is loaded.

4. The LLM is called and asked to return structured JSON.

5. The result is validated and returned.

What the Client Sees:

A block of JSON that includes title, participants, action points, and a summary — ready to embed in a dashboard, export, or tag in a document management system.

Summary Table

Scenario	Mode	Tool(s) Used	Core Value
A. File Download	Standard	`download_file`	Fast and simple access to raw files
B. Secure Transfer	FTI	`secure_transfer`	End-to-end encrypted delivery
C. Metadata Extraction	Standard or FTI	`extract_metadata`	Generative summaries for better search and classification

Notes for Developers

- The three flows all run through the same dispatch pipeline (`tools/call`) but trigger different internal handlers.
- All tools validate schemas up front, including URI normalization and argument checks.
- Tools are discoverable via `tools/list`, with FTI tools hidden unless `fti_mode` is active.

Notes for Stakeholders

- These scenarios can be audited: every step logs API key usage, file access, and time.
- Metadata extraction can run locally (LLM on-prem via Ollama) or via cloud APIs, offering cost and compliance flexibility.
- Secure transfers enforce zero-trust principles — even if TLS is breached, files are unreadable to unauthorized clients.

Failure-Mode Playbook

Below are **six common error scenarios** you can reproduce in a dev environment.
For each case you'll see:

1. **Trigger** – what to change in the request.
2. **Observed response** – HTTP status + JSON-RPC error.
3. **Connector behaviour** – what the code does, which table(s) update.
4. **Why it matters** – security or operability angle.

F1 — Expired or Deactivated API Key

Trigger
Client sends a request using an API key that has either:

- Been marked as inactive (`active = false`), or
- Passed its `expires_at` timestamp.

Expected Response

```
HTTP 403 Forbidden
```

```
{
  "jsonrpc": "2.0",
  "error": {
    "code": -32602,
    "message": "Invalid or expired API key"
  }
}
```

Connector Behavior

The check occurs before any route logic is touched:

- The FastAPI dependency in `router/mcp_router.py` (or `router/admin.py`, etc.) uses the shared guard logic.

It delegates to:

```
services.key_service.KeyService.get_valid_key(api_key: str)
```

That service uses:

```
db.api_key_repo.get_active_key(key)
```

- to query the `user_api_keys` table and verify:
    - `active == True`
    - `expires_at` is not in the past

If the key is invalid, it raises `HTTPException(403)` with JSON-RPC–style error details.

Database Side-Effect

Even though the request is blocked:

- A row **is still inserted** into `access_logs` (via `trace_service.py`)
- But the `api_key_id` is set to `NULL`
- This ensures traceability of unauthorized or expired access attempts

Example log entry:

```
{
  "endpoint": "/mcp",
  "api_key_id": null,
  "timestamp": "2025-07-22T15:03:14Z",
  "ip_address": "127.0.0.1"
}
```

Why It Matters

This provides airtight protection against the re-use of “zombie” credentials:

- Revoked staff tokens
- Expired contractor keys
- Accidentally committed or leaked keys
- Shadow copies from browser caches

It implements **zero trust at the entry point** — the system does not assume any key is safe unless it actively checks every time.

Operational Tip

You can list failed access attempts like this:

```
SELECT * FROM access_logs
WHERE api_key_id IS NULL
ORDER BY timestamp DESC;
```

Use it to:

- Detect scans or unauthorized retries
- Investigate complaints about “blocked” clients
- Confirm key expiry enforcement is working

Cross-Reference

Component	Role
`services/key_service.py`	Top-level logic to validate key

`db/api_key_repo.py`	Fetches active + non-expired keys
`router/mcp_router.py`	Injects API key into route dependencies
`db/models.py`	Defines `user_api_keys` model
`services/trace_service.py`	Logs rejected request to `access_logs`

Summary

This failure mode confirms the connector enforces **up-front rejection** of invalid credentials:

- No session is created
- No tool is invoked
- No access to anything
- Logs are still captured

The system remains inert and untouchable when bad keys are used — a foundational aspect of zero-trust design.

F2 — Role Mismatch: Standard Key Requests FTI Mode

Trigger
Client attempts to initialize an MCP session with:

```
{
  "method": "initialize",
```

```
  "params": { "fti": true }
}
```

using an API key whose role is `"standard"` (instead of `"fti"`).

Expected Response

```
HTTP 200 OK
```

```
{
  "jsonrpc": "2.0",
  "error": {
    "code": -32602,
    "message": "FTI mode not permitted for this key"
  }
}
```

Connector Behavior

This check occurs in:

```
services/mcp_service.py → initialize_result()
```

When `fti: true` is requested:

```
if fti and user.role != "fti":
    raise HTTPException(
        status_code=200,
        detail={"code": -32602, "message": "FTI mode not permitted for
this key"}
    )
```

- The call **does not create a session**
- It **logs the attempt** via `trace_service.py`, so the access trace is still recorded

Database Side-Effect

No row is created in `mcp_sessions`.

But a row **is inserted** into `access_logs`:

```
INSERT INTO access_logs (api_key_id, endpoint, json_body, ...) VALUES
(...)
```

This ensures auditability even for failed escalations.

Why It Matters

This enforces **privilege boundaries** — no standard client can self-upgrade to use sensitive FTI tools (e.g. `secure_transfer`, `rsa_key_exchange`) just by tweaking the `initialize` payload.

It prevents:

- LLMs from hallucinating unauthorized session modes
- Attackers from requesting encryption features by faking session params
- Clients from guessing that "FTI mode" exists and trying to brute-force it

Operator Tip

To audit misuse attempts:

```
SELECT * FROM access_logs
WHERE endpoint = '/mcp'
  AND json_body::text ILIKE '%"fti": true%'
  AND api_key_id IN (
    SELECT id FROM user_api_keys WHERE role != 'fti'
  )
  ORDER BY timestamp DESC;
```

This reveals users who tried to exceed their privileges.

Cross-Reference

Component	Role

`services/mcp_service.py`	Session initialization logic
`db/session_repo.py`	Not called in this failure path
`db/api_key_repo.py`	Provides key + role info
`router/mcp_router.py`	Routes `initialize` calls
`services/trace_service.py`	Logs all attempts
`mcp_sessions` table	Skipped on failure

Summary

- Clients **must use a key with `role = "fti"`** to enable `fti: true` session mode.
- Standard keys are **politely refused**, not blocked.
- This upholds role-based access control and ensures **encryption tools remain protected**.

F3 — Rate-Limit Exceeded: Too Many Requests from the Same API Key

What Triggers This?

If a client sends **too many requests in a short time window**, the server returns a `429 Too Many Requests` error. This protects the backend from overload and discourages abuse.

Default rule for Basic tier:

- Limit: **5 requests per 60 seconds**
- Exceeding this returns:

```
HTTP 429 Too Many Requests
```

```
{
  "jsonrpc": "2.0",
  "error": {
    "code": -32602,
    "message": "Rate limit exceeded"
  }
}
```

How Rate Limiting Works Internally

The FTI-MCP connector uses a **Redis-backed rate limiter** to track usage by each client. This logic lives in:

File	Role
`services/redis_rate_limiter.py`	Implements the Redis rate limiter
`core/limiter.py`	Exposes a singleton `rate_limiter` and `dynamic_rate_limit()` helper
`main.py`	Middleware that calls `dynamic_rate_limit()` before handling each request
`interfaces/rate_limiter_interface.py`	Abstracts the rate limiter contract

The check is done **per user** based on the API key's tier.

The Step-by-Step Flow

Step	Component	Action
1	Client	Sends API request (e.g., `tools/call`)
2	Middleware (`main.py`)	Extracts API key and authenticates
3	`dynamic_rate_limit(request, user)`	Checks if the rate limit was already enforced
4	Redis	Looks up current hit count using key: `rate_limit:{user_id}`
5	If first request	Sets Redis key with TTL = window (e.g., 60 seconds)
6	If limit exceeded	Raises `HTTPException(429, "Rate limit exceeded")`
7	If allowed	Increments counter and continues processing

Sample Code from `redis_rate_limiter.py`

```
cur = await self._redis.get(key)
if cur is None:
```

```
    # First hit → set with expiration
    async with self._redis.pipeline() as pipe:
        pipe.set(key, 1, ex=window)
        await pipe.execute()
elif int(cur) >= max_calls:
    raise HTTPException(429, "Rate limit exceeded")
else:
    await self._redis.incr(key)
```

Why Redis?

- **Atomic counters** — fast and reliable
- **Auto-expiry** — no manual cleanup
- **Horizontal scalability** — ideal for containerized or cloud deployments

Configurable Limits by Tier

Defined in `redis_rate_limiter.py`:

```
_LIMITS = {
    "basic": (5, 60),
    "premium": (30, 60),
    "unlimited": None,
}
```

You can assign tiers via `rate_limit_tier` on the user object in your API key database.

Logging & Traceability

Even failed requests are recorded.

Source	Logged

`access_logs` DB table	Endpoint, API key, IP, timestamp
`/trace` endpoint	View recent activity, including rate limit rejections

This helps in:

- Monitoring abuse
- Troubleshooting false positives
- Analyzing client usage patterns

Exemptions and Overrides

- **Admin keys** and **FTI-mode keys** typically have tier `"unlimited"` and are never rate-limited.
- To simulate exemption during testing, assign `rate_limit_tier = "unlimited"` to your test key.

Developer Tips

Spike-safe: Use Redis scan:

```
redis-cli --scan --pattern "rate_limit:*"
```

Observe abuse:

```
SELECT api_key, COUNT(*)
FROM access_logs
WHERE timestamp > NOW() - INTERVAL '5 minutes'
GROUP BY api_key
ORDER BY count DESC;
```

Test your tier:
Send 6 requests rapidly and observe if the 6th fails with `429`.

Summary

- Rate limits prevent runaway usage or bot abuse.
- Implemented using Redis token-bucket logic.
- Enforced **per user**, not per IP (safer for NAT/shared networks).
- Applies to key routes like `/mcp`, `/tools/call`, `/prompts/run`, etc.
- All activity is logged and auditable via `/trace`.

F4 — Tampered Ciphertext: Integrity Failure in Secure Transfer

What This Means

This error occurs when a recipient tries to decrypt a file **but the ciphertext has been tampered with** — even slightly. It may signal:

- A transmission corruption
- A proxy or malware modifying the payload
- A mismatched key or incorrect decryption logic
- A developer test using incomplete or random ciphertext

In all cases, decryption fails **securely** — the system refuses to return incorrect or dangerous output.

What the Client Sees

During FTI-mode decryption using `secure_transfer`, a failure like this returns:

```
{
  "jsonrpc": "2.0",
  "error": {
    "code": -32001,
    "message": "Decryption failed: invalid ciphertext or integrity check
failed"
  }
```

```
}
```

This happens **before** the file is ever reconstructed on disk.

How FTI Encryption Ensures Integrity

The FTI connector uses **AES-GCM**, a modern encryption mode that:

- Encrypts and authenticates in one pass
- Detects any bit-flip or tampering
- Uses a `tag` (like a cryptographic checksum)

If the `tag` doesn't match the computed value during decryption, it halts with an error.

```
aesgcm = AESGCM(symmetric_key)
plaintext = aesgcm.decrypt(nonce, ciphertext, associated_data=None)
```

If any of:

```
ciphertext_b64
nonce_b64
tag_b64
```

has been altered, decryption will raise a `cryptography.exceptions.InvalidTag` exception internally — and this is converted to the JSON-RPC error above.

Internal Protection Flow

Phase	Module	Action

File transfer	`secure_file_service.py`	AES-GCM + RSA encryption produces ciphertext + tag
File decryption	`decrypt_payload_with_rsa()`	Uses AESGCM.decrypt()
Failure trigger	`InvalidTag` exception	Raised if ciphertext or tag is tampered
Final result	JSON-RPC error	`code -32001` and clear message

Security Implications

✅ **Tamper-Proof Guarantee**
If even 1 bit is changed — due to MITM, proxy, disk error, or corruption — the decryption fails completely.

✅ **Safe Failure**
There's no partial decryption or guessing — attackers cannot infer anything about the original data.

✅ **Zero-Trust Ready**
Even untrusted storage (cloud, email, S3) can carry the payload — it's unreadable without the exact private key and original tag.

What Causes This in Practice?

Cause	Description
Dev test with fake data	Copy-pasted or truncated `ciphertext_b64`
Storage corruption	Damaged file during transit or disk read

Wrong pub/priv key pair	File was encrypted for Bob but Alice is decrypting
Manual editing	Someone altered a base64 field before POSTing
Non-FTI client	A client without AES-GCM support tried to decrypt

Logs & Audit Trail

The failure is captured in:

Channel	Info Logged
`access_logs` table	Timestamp, file attempted, user, API key
`/trace` endpoint	Request payload (redacted), error message
Server stderr (DEBUG)	`cryptography.exceptions.InvalidTag` stack trace

This helps admins verify:

- **Who** tried to decrypt
- **What file** was affected
- **Why** it failed (tampering vs wrong key)

For Developers: Common Pitfalls

- Never try to "peek inside" encrypted payloads.

- Use full, untouched values from `secure_transfer` response.
- Do not change spacing or newline characters in base64.
- Ensure the client uses **matching keypair** from `/rsa_keys`.

Summary

Attribute	Value
Integrity Mechanism	AES-GCM tag check
Common Error Code	`-32001`
Recovery Path	Re-request the encrypted file
Safety	No plaintext ever leaks
Ideal Client Behavior	Fail securely and alert the user

F5 — Mismatched Session ID: Invalid or Missing Context

What This Means

This error occurs when a client tries to make a request (like `tools/call` or `resources/list`) **without a valid or active session**.

In the Model Context Protocol (MCP), every interaction must be part of a session — much like a login session. If that session is:

- Missing

- Expired
- Incorrect for the current API key
- Associated with a different mode (e.g., standard vs FTI)

…then the request is rejected with a clear error.

What the Client Sees

The client receives a JSON-RPC error response like:

```
{
  "jsonrpc": "2.0",
  "error": {
    "code": -32000,
    "message": "No active MCP session found. Please initialize first."
  }
}
```

Or:

```
{
  "jsonrpc": "2.0",
  "error": {
    "code": -32600,
    "message": "Invalid MCP session. This session does not match your
current API key or access mode."
  }
}
```

Why MCP Requires Sessions

MCP treats each client interaction as part of a broader “model context,” which:

- Tracks the **mode** (standard or FTI)
- Logs the **tools used** and **files accessed**
- Applies security checks based on API key and session metadata

This helps enforce strict separation between:

- General API use (e.g., public downloads)
- High-security FTI flows (e.g., encryption, key exchange)

No session = no context = blocked.

Internals: How It Works

Step	Component	Behavior
`initialize`	`session_service.py`	Creates session row in `mcp_sessions` DB table
Per request	`guard_mcp` middleware in `main.py`	Extracts session ID from headers
Validation	`session_repo.py`	Ensures session is active, matches API key, and hasn't expired
Failure	Guard rejects call	Returns code `-32000` or `-32600`

Common Causes

Cause	Description
Forgot to call `initialize`	Client made a request without first creating a session

Missing `Mcp-Session-Id` header	Session ID wasn't sent in the HTTP headers
Expired session	Server cleans up old sessions (e.g., after 1 hour of inactivity)
Switched API keys mid-session	Session is bound to original API key and mode
Session created in wrong mode	E.g., initialized with `fti: false`, then used `secure_transfer`

How It Protects You

✅ Prevents tools from running **without knowing who called them**
✅ Separates **high-trust FTI tools** from standard ones
✅ Ensures **session-bound behavior** — no cross-account leakage
✅ Enables **rich audit trails**, since every action is traceable to a session

For Developers: Best Practices

Always start your flow with an `initialize` call like:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "initialize",
  "params": {
    "protocolVersion": "2025-06-18",
    "fti": true
  }
}
```

- Save the session ID returned (from response header `Mcp-Session-Id`)

- Send it with every request like so:

```
POST /mcp
api-key: YOUR_KEY
Mcp-Session-Id: 9f35111a-234c...
```

- Sessions are not global — each API key gets its own context.

Server-Side Logs

Log	Field
`mcp_sessions`	Session ID, API key, active status, FTI flag
`access_logs`	Each request tied to session
`/trace`	Failed requests logged with reason: `session not found`

This ensures **admins can reconstruct session usage and misuse**.

Summary

Attribute	Value
Error Codes	`-32000`, `-32600`

Root Cause	Missing or invalid session
Impact	All tool/resource calls blocked
Fix	Call `initialize`, resend `Mcp-Session-Id`
Use-case	Prevents session spoofing or tool misuse

F6 — Tool Not Available in Current Mode

What This Means

The client tried to invoke a tool (like `secure_transfer`) that isn't allowed in their current session mode.

In the FTI-MCP Connector, each session operates in one of two modes:

Mode	Tool Access
Standard	Only general-purpose tools (e.g., `download_file`, `extract_metadata`)
FTI Mode	FTI-specific tools allowed (e.g., `secure_transfer`)

Attempting to call an FTI-only tool from a standard session results in a rejection.

What the Client Sees

A typical error message returned to the client:

```
{
  "jsonrpc": "2.0",
  "error": {
    "code": -32601,
    "message": "Tool 'secure_transfer' is not available in your current session mode. Initialize with fti: true."
  }
}
```

Why Modes Exist

This dual-mode design provides **security separation and compliance controls**:

- **Standard Mode**: Open, flexible, lower-security. Suitable for partners or automation tools.
- **FTI Mode**: Locked down, secure, audit-heavy. Required for regulated data transfers.

The system enforces tool visibility based on session mode to **prevent misuse of sensitive features**.

Internals: How It Works

Step	Component	Behavior
`tools/list`	`tool_registry_service.py`	Filters out FTI-only tools if session isn't FTI
`tools/call`	`tool_service.py`	Verifies the requested tool is allowed for this session
Registry	`db.tool_repo.py`	Each tool has an `fti_only` flag in DB

Failure	Middleware raises `-32601`	Client gets a clear rejection with reason

Common Causes

Cause	Example
Called `secure_transfer` in standard mode	Session started without `"fti": true`
Tool was hidden in `tools/list`, but invoked manually	Client used a stale or hardcoded tool list
API key doesn't have `fti` role	Even with `fti: true`, key must have `role: fti`
Session mismatch	Started session with one key, invoked tool with another

How It Protects You

✅ **Blocks unauthorized use** of encryption, key exchange, or secure downloads
✅ Ensures **only high-trust clients** (FTI role) can access critical tools
✅ Prevents **accidental data leaks** by standard sessions calling sensitive logic
✅ Supports **clean capability separation**, making audits easier

Developer Guidance

To access FTI tools like `secure_transfer`, you must:

1. **Have an API key with the `fti` role**
2. **Call `initialize` with `"fti": true`**

Example:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "initialize",
  "params": {
    "protocolVersion": "2025-06-18",
    "fti": true
  }
}
```

If successful, your session will be marked as FTI-mode, unlocking the full toolset.

Tool Visibility & Discovery

- `tools/list` dynamically shows only valid tools for the current session
- FTI-only tools (like `secure_transfer`) are **omitted** unless:
    - Your API key has the FTI role
    - You've initialized with `"fti": true`

This design helps **prevent front-end UIs from leaking advanced tool options** to general users.

Audit Logging

Log	Field

`mcp_sessions`	FTI flag (`true/false`) for each session
`access_logs`	Tool name, API key, session ID, timestamp
`trace`	Failed requests (e.g. tool not allowed) with reason logged

Summary

Attribute	Value
Error Code	`-32601`
Cause	Tool not exposed in current session mode
Fix	Initialize session with `"fti": true` and proper API key
Who Gets It	Clients trying to call FTI tools from standard mode
Benefit	Strong tool isolation by mode, clear security boundaries

F7 — Invalid Tool Arguments

What This Means

A client tried to invoke a tool (e.g. `secure_transfer`, `download_file`) via the MCP `tools/call` method, but passed **invalid arguments** — either:

- A **required field was missing**
- A field was **of the wrong type**
- A **disallowed value** (e.g. unknown action)
- An **unrecognized extra field** was included
- Or a **remote host/port** combo was incorrect or unreachable

Real Examples of Incorrect Tool Calls

❌ **Missing required field: `action`**

```
{
  "name": "secure_transfer",
  "arguments": {
    "filename": "Meeting_1.pdf"
    // 'action' is missing
  }
}
```

❌ **Wrong field type: `remote_port` as string**

```
{
  "name": "secure_transfer",
  "arguments": {
    "action": "download",
    "file_id": "abc123",
    "remote_host": "backend-provider",
    "remote_port": "not-a-number"
  }
}
```

❌ **Invalid action value**

```
{
  "name": "secure_transfer",
  "arguments": {
    "action": "delete",
    "filename": "Meeting_1.pdf"
  }
}
```

What the Client Sees

A structured JSON-RPC error like:

```
{
  "jsonrpc": "2.0",
  "id": 4,
  "error": {
    "code": -32602,
    "message": "Invalid arguments for tool 'secure_transfer': missing
required property 'action'"
  }
}
```

Or if Pydantic fails:

```
{
  "code": -32602,
  "message": "Expected 'remote_port' to be integer"
}
```

Internals: How Validation Works

Every tool has an **input schema** defined as a `dict`, either manually or via Pydantic, like:

`secure_transfer` (simplified schema logic):

```
{
  "action": {"type": "string"},
```

```
  "filename": {"type": "string"},
  "description": {"type": "string"},
  "file_id": {"type": "string"},
  "remote_host": {"type": "string"},
  "remote_port": {"type": "integer"},
  ...
}
```

The exact required fields vary **depending on the `action`** (`upload`, `download`, `decrypt`), so handlers **must validate combinations dynamically**.

Validation occurs inside:

- `handlers/secure_transfer_tool.py`
- `handlers/download_file.py`
- `tool_service.py` (`call_tool()`)

Developer Notes

Tip	Why it helps
Use `tools/list`	See the expected input schema before calling a tool
Validate conditional fields	Some arguments only make sense for a certain action
Always pass the `action`	It is the entry point for branching logic
Use `file_id`, `remote_host`, `remote_port` correctly	For remote file pulls, all three must be valid and present
Check file existence for `download_file`	The filename must resolve using `resolve_path()`

Correct Usage Examples

Upload a file

```
{
  "name": "secure_transfer",
  "arguments": {
    "action": "upload",
    "filename": "Meeting_1.pdf",
    "description": "Daily trading P&L report"
  }
}
```

Download a file from remote provider

```
{
  "name": "secure_transfer",
  "arguments": {
    "action": "download",
    "file_id": "2edcc454ea4741f2b51ad96f2e5237cc",
    "remote_host": "backend-provider",
    "remote_port": 8000
  }
}
```

Decrypt on consumer

```
{
  "name": "secure_transfer",
  "arguments": {
    "action": "decrypt",
    "key_id": 2,
    "enc_sym_key_b64": "...",
    "nonce_b64": "...",
    "ciphertext_b64": "...",
    "sha256_b64": "..."
  }
}
```

Download local file

```
{
  "name": "download_file",
```

```
  "arguments": {
    "uri": "Meeting_1.pdf"
  }
}
```

Logs and Trace Capture

Invalid tool calls are recorded in trace logs:

```
{
  "tool": "secure_transfer",
  "error": "Invalid arguments: missing 'action'",
  "timestamp": "...",
  "api_key": "...",
  "session_id": "..."
}
```

Summary

Field	Value
Code	`-32602`
Root Cause	Bad/missing/invalid arguments
Triggers	Misuse of `tools/call` with malformed `params.arguments`
Impact	Tool is not called; no side-effects
Logging	Captured in `trace_service` and JSON-RPC error
Prevention	Check schemas via `tools/list` and follow conditional logic in handlers

Tools Affected	All — especially `secure_transfer`, `download_file`, `extract_metadata`

Deployment & Operations

Containerization & Orchestration

This section explains how the full MCP connector system — including the backend APIs, databases, and Redis — is packaged, composed, and run using Docker and Docker Compose. The goal is to make the environment easy to reproduce, maintain, and deploy.

It covers two deployment modes:

- **Single-backend** mode (for local dev)
- **Dual-backend (Provider–Consumer)** mode for full FTI-style cross-server testing

Purpose

Whether you're a developer, DevOps engineer, or project manager, understanding containerization lets you:

- Quickly spin up the full system on any machine
- Avoid "works on my machine" issues
- Confidently replicate the environment in CI/CD or production
- Understand how backends and dependencies (like Redis/PostgreSQL) interact

Component Overview

Service	Role	Image or Source

`backend-provider`	Provider backend exposing `/mcp`, tools, FTI	Built from local Dockerfile
`backend-consumer`	Consumer backend pulling from remote provider	Built from local Dockerfile
`db-provider`	PostgreSQL DB for provider backend	`postgres:15`
`db-consumer`	PostgreSQL DB for consumer backend	`postgres:15`
`redis`	Shared Redis instance	`redis:7-alpine`

Each component runs as an isolated container. This modularity ensures versioned consistency and easier debugging or scaling.

Dockerfile (for both backends)

```
FROM python:3.11-slim

WORKDIR /app

RUN apt-get update && \
    apt-get install -y build-essential libpq-dev gcc nodejs npm

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

CMD ["uvicorn", "main:app", "--reload", "--host", "0.0.0.0", "--port",
"8000", "--reload-dir", "/app"]
```

Explanation:

- python:3.11-slim — lightweight Python base image
- libpq-dev — needed by psycopg2 for PostgreSQL
- nodejs, npm — optional; used by frontend if present
- --reload-dir /app — enables live reload when code changes

docker-compose.yml: Dual-Backend Setup

This docker-compose.yml file defines a fully containerized simulation of two distinct FTI-compatible backends — a **provider** and a **consumer** — each operating in its own environment, yet communicating securely over shared infrastructure. It also includes supporting services like **PostgreSQL** (for persistence) and **Redis** (for caching and coordination).

The YAML below is the actual configuration:

```
version: "3.9"

services:
  db-provider:
    image: postgres:15
    restart: unless-stopped
    environment:
      POSTGRES_USER: fti_user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: fti_provider
    ports: [ "5434:5432" ]
    volumes: [ pg_provider:/var/lib/postgresql/data ]

  db-consumer:
    image: postgres:15
    restart: unless-stopped
    environment:
      POSTGRES_USER: fti_user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: fti_consumer
    ports: [ "5435:5432" ]
    volumes: [ pg_consumer:/var/lib/postgresql/data ]

  redis:
```

```
    image: redis:7-alpine
    restart: unless-stopped
    ports: [ "6379:6379" ]

  backend-provider:
    build: .
    env_file: .env.provider
    depends_on: [ db-provider, redis ]
    ports: [ "8000:8000" ]
    volumes:
      - .:/app
      - ./shared_data:/data/provider

  backend-consumer:
    build: .
    env_file: .env.consumer
    depends_on: [ db-consumer, redis, backend-provider ]
    ports: [ "8001:8000" ]
    volumes:
      - .:/app
      - ./shared_data:/data/consumer

volumes:
  pg_provider:
  pg_consumer:
```

Explanation of Key Services & Their Roles

- **db-provider** / **db-consumer**:
  These are two separate PostgreSQL containers, each tied to its respective backend. They simulate independent deployments and store persistent data like users, sessions, API keys, and encrypted file metadata. Each uses a dedicated port (`5434`, `5435`) and persistent named volume (`pg_provider`, `pg_consumer`).

- **redis**:
  A lightweight, in-memory Redis server used by both backends. It supports real-time rate limiting, potential event queues, and future publish/subscribe messaging. The Redis service uses port `6379`, which is the default.

- **backend-provider**:
  This is the main application container for the **provider** role. It builds the FastAPI backend from the local Dockerfile, loads environment-specific config from `.env.provider`, and mounts the local project directory (`.`) for live code sync. It also

mounts `shared_data` for file exchange, but only the `provider` subdirectory.

- **backend-consumer**:
  A second FastAPI backend for the **consumer** role. Like the provider, it builds from the same Dockerfile but loads from `.env.consumer`. It depends on both its own database, Redis, and also the provider (to support remote secure file pulls). Its file mount is mapped to `shared_data:/data/consumer`.

Key Concepts & Best Practices

Directive	Explanation
`restart: unless-stopped`	Ensures that containers automatically restart after a system reboot, unless explicitly shut down by the user.
`depends_on`	Guarantees startup order by ensuring that backends launch only after their database and Redis are up and accepting connections. Prevents boot errors.
`ports`	Maps internal container ports to host ports. For example, the provider backend is accessible at `http://localhost:8000` and the consumer at `http://localhost:8001`.
`env_file`	Injects configuration values (like database URLs, API secrets, or Redis URIs) from a separate `.env` file. Keeps secrets out of source code.
`volumes` (named)	Used for persisting PostgreSQL data even if containers are rebuilt. Named volumes like `pg_provider` ensure long-term data retention.

`volumes` (bind mount)	Mounts the local project directory into the container so that file changes on the host are instantly visible to the container. Boosts dev velocity.
`build: .`	Builds the backend images directly from the local Dockerfile. Ensures consistent FastAPI code across both provider and consumer instances.

Why This Setup?

This structure reflects a secure multi-party communication model where:

- Each backend can operate independently with its own database and environment.
- Redis serves as a shared, stateless coordination layer.
- File transfers and metadata queries happen over authenticated HTTP.
- Realistic FTI scenarios (e.g., remote access, digital signatures, cross-backend validation) can be tested locally with full isolation and observability.

Common Commands

Action	Command	Notes
Start all containers	`docker compose up --build`	Fresh build + boot
Start (no rebuild)	`docker compose up`	Use when only `.env` or mounts changed
Tear down (keep data)	`docker compose down`	Leaves volumes intact

Tear down (delete data)	`docker compose down -v`	Deletes PostgreSQL volumes
Shell into provider backend	`docker compose exec backend-provider bash`	Useful for DB access or migrations
Shell into consumer backend	`docker compose exec backend-consumer bash`	Same as above

Production Guidance

Area	Dev Setting	Production Best Practice
Entrypoint	`uvicorn` + `--reload`	Use Gunicorn (`gunicorn -w 4 -k uvicorn.workers.UvicornWorker main:app`)
Secrets	`.env` file	Use Docker/K8s secrets or vaults
Backups	Manual `pg_dump`	Scheduled automated backups to cloud
Logs	Terminal stdout	Centralized logging (ELK, Loki, etc.)
Scaling	Single containers	Use Docker Swarm, Kubernetes, ECS, or similar orchestrator

Static files	Served by FastAPI	Serve via Nginx or a CDN

Troubleshooting

Symptom	Likely Cause	How to Check
API container exits immediately	Python error, missing var	`docker compose logs backend-provider`
Cannot connect to DB	DB not ready yet	Check `depends_on` and wait for "ready to accept" logs
Port already in use	Conflict with local service	`lsof -i :8000` or `netstat -a -b -n` (Win)
Image rebuild takes forever	Cache busting (e.g. `requirements.txt` change)	Try keeping unchanged layers near the top of Dockerfile

Summary

- The entire MCP connector stack is reproducibly containerized.
- With two commands, any developer or operator can boot the full environment.
- Docker Compose cleanly defines the topology: databases, cache, backends.
- Transitioning to production is a matter of refining the entrypoint and externalizing secrets/logs.

Extensibility & Roadmap

Overview

The FTI-Compatible MCP Connector is designed for extensibility — not just as a software best practice, but as a foundational stance on how modern systems should evolve. Whether you're a public-sector integrator, a compliance-minded enterprise, or a developer experimenting with secure automation, this connector can adapt to your infrastructure without forcing vendor lock-in or rigid assumptions.

Its modular internals — covering everything from authentication and file encryption to tool execution and LLM use — are designed to be replaceable. The system runs cleanly even in minimal setups, but can also scale up with Redis Streams, OAuth2, cloud storage, or policy-aware event chains.

This section details:

- What you can extend or replace today.
- How breaking changes are managed.
- What’s planned over the next 12 months.
- How to contribute.
- And our long-term vision for secure, decentralized infrastructure.

Extension Points (What Can Be Customized Today)

Each row below describes a "plug zone" — a part of the connector that's intentionally swappable via interfaces or environment settings.

Area	Current Default Behavior	How to Extend or Replace It
Authentication	API key checked against PostgreSQL (`api_key_repo.py`)	Replace with OAuth2, mTLS, or workload identity (e.g., SPIFFE). Shared interface ready.

Resource Storage	Files stored under `./shared_data` (`resource_service.py`)	Swap in Amazon S3, Azure Blob, MinIO, or even a remote MCP over HTTP. URI-scheme aware.
Event Bus	In-memory publish/subscribe system (per node only)	Swap for Redis Streams or NATS for multi-node coordination. Protocol stays unchanged.
Encryption	Hybrid RSA-2048 + AES-GCM using public keys from `rsa_keys` table	Plug in post-quantum crypto or HSM-backed key wrapping by modifying `encryption_utils.py`.
Tools	Decorated Python callables (e.g., `@tool_registry.register`)	Add or modify tools by simply writing a new function + annotation, then restart the server.
LLM Integration	Gemini 1.5 with fallback to DeepSeek (via `llm_service.py`)	Easily swap in OpenAI, Claude, Cohere, or local Ollama via adapter pattern.
Rate Limiting	Redis-backed fixed window (e.g., 5 req/min per API key tier)	Replace with token buckets or exponential backoff inside `redis_rate_limiter.py`.
Audit Logging	Recorded to PostgreSQL (`trace_repo.py`)	Replace with Kafka, JSONL files, or a Merkle-chained ledger with cryptographic signatures.

All of these areas are interface-driven and isolated by module — swapping behavior never requires changes to the rest of the system.

Breaking-Change Policy

Stable ecosystems need predictable evolution. The connector adopts strict policies to ensure backwards compatibility and migration support.

Layer	Policy
MCP Protocol	Versioned via `protocolVersion`. Older clients supported for 6 months.
HTTP Routes	Never renamed or removed. Deprecated routes emit `X-Deprecated` header.
Database Schema	All changes tracked via Alembic with downgrade scripts if needed.
Environment Vars	All new vars have safe defaults. Breaking flags never required blindly.

In short: you won't get blindsided. You'll always have a clear upgrade path.

One-Year Roadmap

Q3 2025 – Resilience & Scaling

- ✅ Redis Streams for multi-node `tools/list_changed` and `resources/updated` event broadcasting.
- ✅ `/metrics` endpoint with Prometheus-compatible counters and rates.
- ✅ Admin-triggered RSA key rotation with live rollout (no downtime).

Q4 2025 – Policy & Compliance

- IDSA-compliant usage contracts embedded into tool and resource metadata.

- Cryptographically chained audit logs (Merkle trees anchored in public ledgers).
- Hardened Docker images (CIS Level 1), plus cloud deployment templates for AWS/GCP/Azure.

Q1 2026 – Storage & Virtualization

- Native support for presigned file uploads and reads from S3, Azure Blob, or GCS.
- Support for “virtual resources” (e.g., SQL queries or metadata JSON objects as URI targets).

Q2 2026 – Smart Automation & Local AI

- Trigger actions (e.g., `extract_metadata`, `notify`) when new files arrive.
- Native support for on-premise LLMs like Ollama with tuning profiles and security sandbox.
- Launch a lightweight reference UI for drag-and-drop uploads, metadata editing, and report export.

How to Contribute

The project is open-source, interface-oriented, and easy to extend. Here's how to get started:

- **Fork → Feature Branch → Pull Request**: GitHub flow with reviews.
- **Style**: `black` and `ruff`, enforced by `pre-commit`.
- **Tests**: `pytest` + SQLite + fake LLM/adapters. No cloud calls in CI.
- **CI/CD**: GitHub Actions build/test; merge to `main` → pushed Docker image (`:edge`).
- **Security Review**: All crypto, auth, and session-related changes must include a threat note in `SECURITY.md`.

Vision: Where This Is Headed

This connector isn't just a server — it's a **trust protocol**, an enabler of collaboration across organizational boundaries without sacrificing sovereignty.

Federation by Default

Organizations can run their own connectors and share **tools**, **prompts**, and **resource metadata** via controlled protocols — no central broker required.

Attribute-Based Encryption

Encrypt portions of a file, not just the whole thing. PDFs or JSON objects could reveal sections based on user clearance, project ID, or time of day.

Zero-Trust by Design

API keys are just the beginning. Future plans include support for **ephemeral SPIFFE identities**, **workload attestation**, and **mutual TLS with certificate pinning**.

Edge Computing

Users may upload logic — e.g., a WASM or Python snippet — to process their data *on arrival*. The server validates and runs it in a sandbox, extracting insight while maintaining trust boundaries.

Final Thought

The MCP Connector today is a secure, extensible backend for controlled file exchange and metadata automation. Tomorrow, it's the foundation for a distributed, programmable trust fabric.

It's built not just to work — but to **evolve**.

Appendix

This appendix is a comprehensive reference for developers, DevOps engineers, and project stakeholders. It consolidates key configuration variables, public APIs, protocol behaviors, and glossary terms to support deployment, troubleshooting, and extension of the FTI-Compatible MCP Connector.

Environment Variables Reference

The connector loads configuration from environment variables, typically declared in `.env.provider` or `.env.consumer` depending on the role. These control authentication, encryption, LLMs, event subscriptions, and runtime behavior.

Variable	Default	Required	Purpose
`DATABASE_URL`	`postgresql+psycopg2://fti_user:password@localhost:5432/fti_project`	✅ Yes	PostgreSQL connection for the local backend instance.
`MASTER_KEY`	none	✅ Yes	32-byte base64 key used to encrypt RSA private keys at rest.
`GEMINI_API_KEY`	empty	❌ No	Enables Google Gemini 1.5-based LLM metadata extraction.
`DEEPSEEK_API_KEY`	empty	❌ No	Optional LLM fallback if Gemini fails.
`AUTO_METADATA`	`true`	❌ No	Automatically extracts metadata on `resources/read`.

`OLLAMA_ENDPOINT`	`http://localhost:11434/api/chat`	❌ No	Endpoint for optional local LLM (e.g., Qwen/Ollama).
`OLLAMA_MODEL`	`hhao/qwen2.5-coder-tools:1.5b`	❌ No	Local LLM image used for metadata, prompt testing, or fallback.
`REQUIRE_HTTPS`	`true`	❌ No	Enforces HTTPS-only traffic in production.
`REDIS_URL`	`redis://redis:6379`	❌ No	Used for Redis-backed rate limiting.
`KEY_RETENTION_DAYS`	`90`	❌ No	Number of days old RSA keys remain valid for decryption.
`CORS_ORIGINS`	`*`	❌ No	Comma-separated list of frontend domains allowed by CORS.
`RES_SUBSCRIBE_ENABLED`	`false`	❌ No	Enables SSE subscriptions for `resources/u`

			`pdated` events.
`TOOLS_LIST_CHANGED_ENABLED`	`false`	❌ No	Live tool registry updates via SSE.
`PROMPTS_LIST_CHANGED_ENABLED`	`false`	❌ No	Enables prompt registry change notifications via SSE.
`PORT`	`8000`	❌ No	Port Uvicorn binds to for the FastAPI app.

Copy a starter `.env` file from `utils.env`, then customize as needed.

API Reference

REST Endpoints

These endpoints provide admin functionality, encryption utilities, and a bootstrap interface for external clients.

Path	Method	Description	Auth Required
`/admin-key`	POST	One-time bootstrap for first admin API key	No

`/keys/create`	POST	Creates a new API key with optional roles	Admin
`/keys/list`	GET	Lists all API keys and their limits	Admin
`/keys/deactivate`	POST	Deactivates an API key	Admin
`/generate-keys`	POST	Generates a new RSA key pair	Any
`/keys/latest`	GET	Fetches current public RSA key (encryption)	Any
`/download`	GET	Streams raw file from a local `file://` URI	Any
`/trace`	GET	View audit logs from access table	Admin
`/mcp`	POST	JSON-RPC endpoint for structured operations	Any
`/mcp/stream`	GET (SSE)	Live updates: resources, tools, prompts	Any

Explore `/docs` or `/redoc` while the server is running.

JSON-RPC Methods

The `/mcp` endpoint accepts all JSON-RPC 2.0 methods. Input and output follow a consistent envelope format.

Session Management

- `initialize(protocolVersion, fti?)` → Negotiates MCP version, returns session headers
- `notifications/initialized()` → Server receives but does not respond (fire-and-forget)

Resources

- `resources/list(cursor?, limit?)` → Returns paginated resource metadata
- `resources/read(uri)` → Fetches full content and extracted metadata
- `resources/subscribe(uri)`, `resources/unsubscribe(uri)`

Tools

- `tools/list()` → All available tools and schemas
- `tools/call(name, args)` → Executes selected tool

Prompts

- `prompts/list()`, `prompts/get(name)` → Retrieve and use prompt templates
- `prompts/run(name, args)` → Run LLM with prompt scaffolding

Notifications

- `notifications/resources/updated(uri)` → SSE event on resource updates
- `notifications/tools/list_changed()` → Tool registry SSE change push

Error Format

```
{
  "jsonrpc": "2.0",
```

```
  "id": "abc123",
  "error": {
    "code": -32602,
    "message": "Rate limit exceeded"
  }
}
```

Glossary

Term	Meaning
MCP	Media Context Protocol: standard JSON-RPC interface for data & tools
FTI Mode	Enhanced security mode used for key-wrapped encrypted transfers
API Key	43-character token, bound to a role (admin, fti, etc.)
Session	Temporary stateful context, initiated via `initialize`
Tool	Callable function exposed over JSON-RPC (`tools/call`)
Hybrid Encryption	RSA (asymmetric) wraps AES-GCM (symmetric) keys
Nonce	Unique 12-byte IV for AES-GCM
Tag	AES-GCM authentication tag (ensures file was not tampered)

TTL	Time-to-live for Redis counters, e.g., in rate limiting
SSE	Server-Sent Events for real-time client-side notifications
Prompt	Reusable input format for LLMs (used in `prompts/run`)

Closing Summary

- The system now supports **dual-backend deployment** (provider and consumer), each with its own DB and secure file lifecycle.
- All components are containerized via **Docker Compose**, and support Redis-backed rate limiting, audit logging, and session-based access control.

Next Steps

For **Developers & Integrators**:

- Use `.env` to bootstrap your local setup
- Explore `/http_tester/` and test via `tools/list` and `tools/call`
- Use the dual backend setup to simulate secure file exchange

For **Admins & Operators**:

- Monitor `/trace`, test rate limits, rotate keys
- Configure your environment with `.env.provider` and `.env.consumer`

For **Security Teams**:

- Review Section 9.4 (Crypto)
- Validate signed upload flow via `/mcp/receive-file`

- Audit key and file lifecycle logs

For **Decision Makers**:

- Check Section 12.1 for all extension points
- Consult the roadmap (12.3) to align features with business timelines
- Consider the upcoming web-based UI for broader internal adoption