Understanding Model Context Protocol (MCP)
MCP Learning Path: From Zero to Hero
Quickstart
Core Concepts
MCP Architecture: Design Philosophy & Engineering Principles
Prompts
Resources
Roots
Sampling
Tools
Transports
MCP Tutorials: From Concept to Production
MCP Server Ecosystem: From Proof-of-Concept to Production
Clients
MCP FAQ: Expert Answers to Real-World Questions
MCP Best Practices: Architecture & Implementation Guide
SDK
Build an MCP Client (Core)
More
Docs ↗
Specification ↗
About

MCP Architecture: Design Philosophy & Engineering Principles

MCP Architecture: Design Philosophy & Engineering Principles

Understanding MCP's architecture requires thinking beyond simple client-server patterns. This is a **protocol designed for AI-first computing**, where traditional request-response models meet the dynamic, context-rich world of Large Language Models.

> 🏗️ **Architectural Perspective**: MCP solves the "AI Integration Paradox" - how to give AI systems rich, secure access to external resources without creating security nightmares or integration complexity.

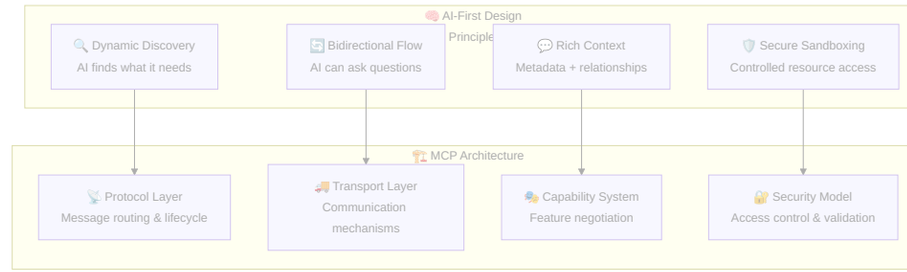
🎯 Design Philosophy: Why These Choices Matter

The AI Integration Challenge

Traditional APIs were designed for **predictable, human-designed workflows**. AI systems need:

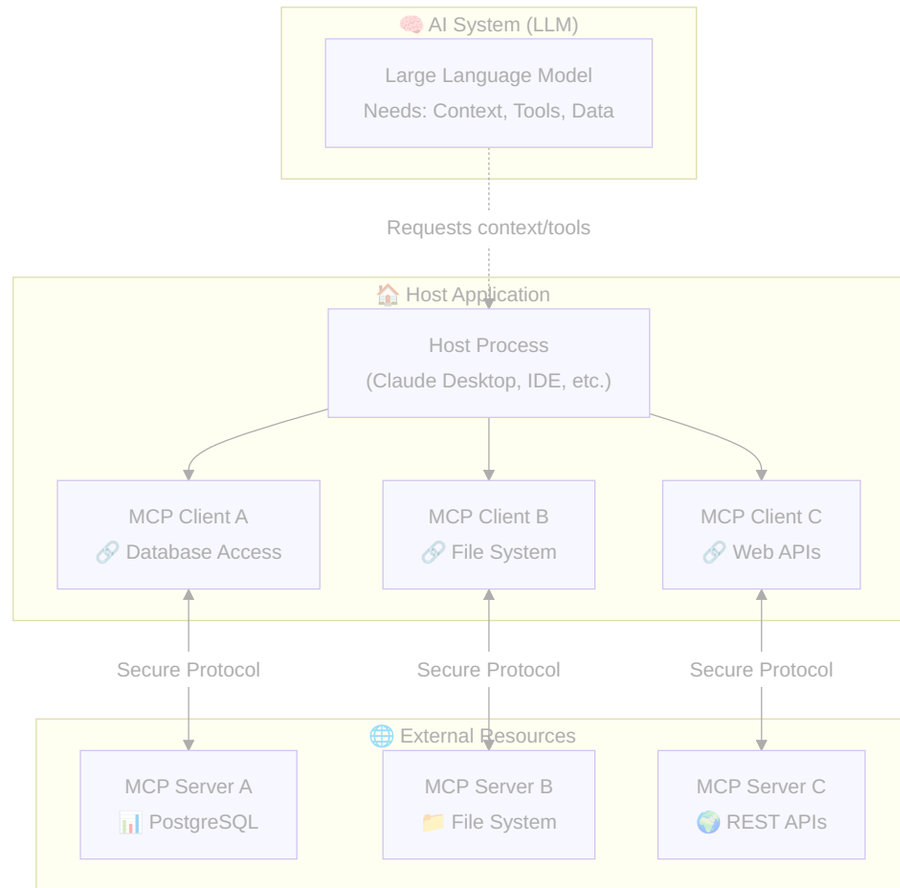
- **Dynamic resource discovery** (AI doesn't know what it needs until it needs it)
- **Rich context exchange** (not just data, but metadata, relationships, capabilities)
- **Secure sandboxing** (AI can't be trusted with direct system access)
- **Bidirectional communication** (AI needs to ask questions, not just consume data)

MCP's Architectural Response



🏛️ Core Architecture: Beyond Client-Server

MCP implements a **"Mediated Access Pattern"** - the Host acts as a security broker between AI and external resources:



🔑 Key Architectural Insights

1. **Host as Security Broker**: The Host mediates ALL AI-resource interactions

2. **1:1 Client-Server Mapping**: Each resource type gets dedicated, isolated communication
3. **Capability-Based Security**: Servers declare what they can do, Hosts decide what to allow
4. **Transport Agnostic**: Protocol works over stdio, HTTP, WebSockets, etc.

🏗️ Layered Architecture: Separation of Concerns

Protocol layer

The protocol layer handles message framing, request/response linking, and high-level communication patterns.

TypeScript Python

```
class Protocol<Request, Notification, Result> {
    // Handle incoming requests
    setRequestHandler<T>(schema: T, handler: (request:

    // Handle incoming notifications
    setNotificationHandler<T>(schema: T, handler: (not

    // Send requests and await responses
    request<T>(request: Request, schema: T, options?:

    // Send one-way notifications
    notification(notification: Notification): Promise<
}
```

Key classes include:

- `Protocol`
- `Client`
- `Server`

Transport layer

The transport layer handles the actual communication between clients and servers. MCP supports multiple transport mechanisms:

1. **Stdio transport**
   - Uses standard input/output for communication
   - Ideal for local processes
2. **HTTP with SSE transport**
   - Uses Server-Sent Events for server-to-client messages
   - HTTP POST for client-to-server messages

All transports use JSON-RPC 2.0 to exchange messages. See the specification for detailed information about the Model Context Protocol message format.

Message types

MCP has these main types of messages:

1. **Requests** expect a response from the other side:

```
interface Request {
  method: string;
  params?: { ... };
}
```

2. **Results** are successful responses to requests:

```
interface Result {
  [key: string]: unknown;
}
```

3. **Errors** indicate that a request failed:

```
interface Error {
  code: number;
  message: string;
  data?: unknown;
}
```

4. **Notifications** are one-way messages that don't expect a response:

Welcome to modelcontextprotocol.info

modelcontextprotocol.info asks for your consent to use your personal data for the following purposes:

Personalised advertising and content, advertising and content measurement, audience research and services development

Store and/or access information on a device

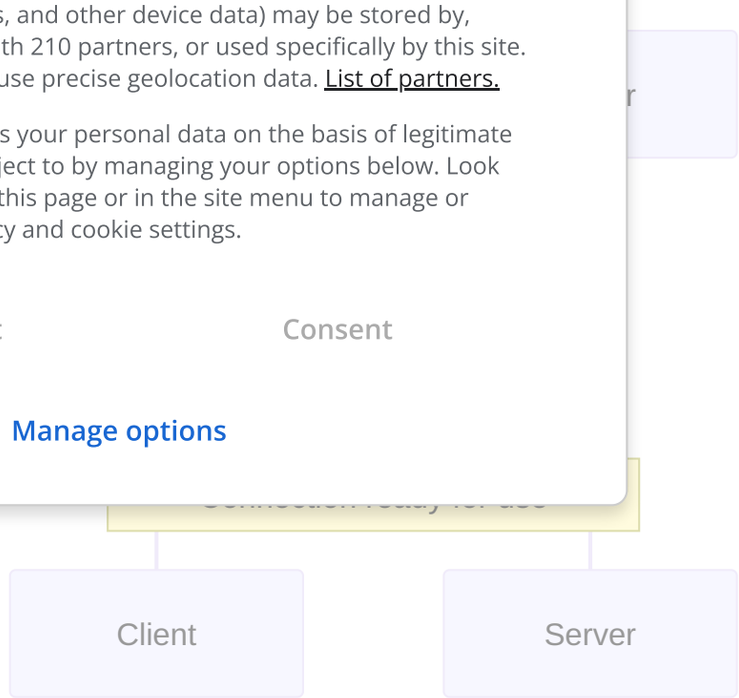
Learn more

Your personal data will be processed and information from your device (cookies, unique identifiers, and other device data) may be stored by, accessed by and shared with 210 partners, or used specifically by this site. We and our partners may use precise geolocation data. List of partners.

Some vendors may process your personal data on the basis of legitimate interest, which you can object to by managing your options below. Look for a link at the bottom of this page or in the site menu to manage or withdraw consent in privacy and cookie settings.

Do not consent Consent

Manage options



1. Client sends `initialize` request with protocol version and capabilities
2. Server responds with its protocol version and capabilities
3. Client sends `initialized` notification as acknowledgment
4. Normal message exchange begins

2. Message exchange

After initialization, the following patterns are supported:

- **Request-Response**: Client or server sends requests, the other responds
- **Notifications**: Either party sends one-way messages

3. Termination

Either party can terminate the connection:

- Clean shutdown via `close()`
- Transport disconnection
- Error conditions

Error handling

MCP defines these standard error codes:

```
enum ErrorCode {
  // Standard JSON-RPC error codes
  ParseError = -32700,
  InvalidRequest = -32600,
  MethodNotFound = -32601,
  InvalidParams = -32602,
  InternalError = -32603
}
```

SDKs and applications can define their own error codes above -32000.

Errors are propagated through:

- Error responses to requests
- Error events on transports
- Protocol-level error handlers

Implementation example

Here's a basic example of implementing an MCP server:

TypeScript Python

```
import { Server } from "@modelcontextprotocol/sdk/serv
import { StdioServerTransport } from "@modelcontextprc

const server = new Server({
  name: "example-server",
  version: "1.0.0"
}, {
  capabilities: {
```

```
    resources: {}
  }
});

// Handle requests
server.setRequestHandler(ListResourcesRequestSchema, a
  return {
    resources: [
      {
        uri: "example://resource",
        name: "Example Resource"
      }
    ]
  };
});

// Connect transport
const transport = new StdioServerTransport();
await server.connect(transport);
```

Best practices

Transport selection

1. **Local communication**
   - Use stdio transport for local processes
   - Efficient for same-machine communication
   - Simple process management
2. **Remote communication**
   - Use SSE for scenarios requiring HTTP compatibility
   - Consider security implications including authentication and authorization

Message handling

1. **Request processing**
   - Validate inputs thoroughly
   - Use type-safe schemas
   - Handle errors gracefully
   - Implement timeouts
2. **Progress reporting**
   - Use progress tokens for long operations

    - Report progress incrementally
    - Include total progress when known
3. **Error management**
    - Use appropriate error codes
    - Include helpful error messages
    - Clean up resources on errors

Security considerations

1. **Transport security**
    - Use TLS for remote connections
    - Validate connection origins
    - Implement authentication when needed
2. **Message validation**
    - Validate all incoming messages
    - Sanitize inputs
    - Check message size limits
    - Verify JSON-RPC format
3. **Resource protection**
    - Implement access controls
    - Validate resource paths
    - Monitor resource usage
    - Rate limit requests
4. **Error handling**
    - Don't leak sensitive information
    - Log security-relevant errors
    - Implement proper cleanup
    - Handle DoS scenarios

Debugging and monitoring

1. **Logging**
    - Log protocol events
    - Track message flow
    - Monitor performance

- Record errors

2. **Diagnostics**
    - Implement health checks
    - Monitor connection state
    - Track resource usage
    - Profile performance
3. **Testing**
    - Test different transports
    - Verify error handling
    - Check edge cases
    - Load test servers

© 2024 ModelContextProtocol.Info.

Welcome to modelcontextprotocol.info

modelcontextprotocol.info asks for your consent to use your personal data for the following purposes:

Personalised advertising and content, advertising and content measurement, audience research and services development

Store and/or access information on a device

Your personal data will be processed and information from your device (cookies, unique identifiers, and other device data) may be stored by, accessed by and shared with 210 partners, or used specifically by this site. We and our partners may use precise geolocation data. List of partners.

Some vendors may process your personal data on the basis of legitimate interest, which you can object to by managing your options below. Look for a link at the bottom of this page or in the site menu to manage or withdraw consent in privacy and cookie settings.