MCP Server Implementation Guide for ServiceNow Integration

Part 5: Appendices and Recommendations

Document Series

This implementation guide is organized into five parts:

1. Part 1: Introduction and Overview
2. Part 2: Server Foundation - Core Infrastructure Setup
3. Part 3: MCP Protocol Implementation and Tools
4. Part 4: OAuth 2.1 Implementation
5. **Part 5: Appendices and Recommendations** (This Document)

Introduction

This document provides supplemental guidance and best practices that complement the core implementation covered in Parts 1-4. These appendices are optional but recommended for production deployments, team-based development, and operational excellence.

What's Included:

- Appendix A: Repository and version control practices
- Appendix B: Alternative storage implementation patterns
- Appendix C: Production deployment checklist
- Appendix D: ServiceNow connection configuration guide
- Reference implementation files documentation

When to Use Appendices:

- Setting up version control for your MCP server project
- Choosing storage solutions (database vs file-based vs Redis)
- Preparing for production deployment
- Configuring ServiceNow MCP client connection
- Understanding reference implementation files

Reference Implementation Files

The following reference files accompany this documentation:

Complete Implementations:

- `server-compliant.js` - Local VM deployment (JavaScript) with file-based client storage, Redis token blacklist, and all documentation compliance features
- `gcp-compliant.ts` - Google Cloud deployment (TypeScript) with Firestore client storage, in-memory token blacklist, and compliance features

Templates:

- `server-example.js` - White-label template with code structure, logging statements, and implementation placeholders
- `.env.example` - Environment configuration template with all required and optional variables documented

Visual Resources:

- `oauth-flow-diagram.mermaid` - OAuth 2.1 flow sequence diagram (Mermaid format for GitHub/GitLab)
- `oauth-flow-diagram.svg` - OAuth 2.1 flow sequence diagram (SVG format for universal compatibility)

Implementation Comparison:

Aspect	server-compliant.js	gcp-compliant.ts
Language	JavaScript	TypeScript
Platform	Local VM, any Node.js environment	Google Cloud Run optimized
Client Storage	File-based (registered_clients.json)	Google Firestore (NoSQL database)
Token Blacklist	Redis with persistence	In-memory Set (acceptable for auto-scaling)
Tools	LLM (Ollama) + File operations	A2A Agent + Utility tools
Use Case	Local development, on-premises	Cloud-native deployments

Both implementations follow the same architectural patterns and are fully ServiceNow-compatible.

Appendix A: Repository Practices (Optional)

This appendix provides guidance for practitioners who want to version control their MCP server implementation using Git or similar version control systems.

Git Ignore Configuration

Create a `.gitignore` file to prevent committing sensitive or generated files:

Essential Exclusions:

- `.env` - Contains secrets and environment-specific configuration
- `node_modules/` - Dependency packages (reinstalled via npm/yarn)
- `*.log` - Log files
- `*.pid` - Process ID files
- `registered_clients.json` - Contains OAuth client secrets

Optional Exclusions:

- `.DS_Store` - macOS system files
- `dist/` or `build/` - Compiled/transpiled code (if using TypeScript)
- `.vscode/` or `.idea/` - IDE configuration (unless shared in team)
- `coverage/` - Test coverage reports

Example .gitignore:

```
# Environment and Secrets
.env
.env.local
.env.*.local

# Dependencies
node_modules/

# Logs
logs/
*.log
npm-debug.log*

# Runtime Data
*.pid
registered_clients.json

# Operating System
.DS_Store
Thumbs.db

# IDE
.vscode/
.idea/
*.swp
*.swo

# Build Output
dist/
build/

# Test Coverage
coverage/
```

README.md Recommendations

A well-structured README helps team members and future maintainers understand your MCP server implementation.

Suggested README Structure:

```
markdown
```

~~~markdown
# MCP Server - [Your Project Name]

Brief description of what this MCP server does and which ServiceNow instance it integrates with.

## Features

- List of implemented MCP tools
- Key capabilities and integrations
- ServiceNow-specific customizations

## Prerequisites

- Node.js v18+ (or your specific version requirement)
- Redis server (if using Redis for token storage)
- ServiceNow instance (Yokohama Patch 9+ or Zurich Patch 2+)

## Setup

1. Clone the repository
2. Copy `.env.example` to `.env` and configure
3. Install dependencies: `npm install`
4. Generate secure secrets (see Security section)
5. Start the server: `npm start`

## Configuration

See `.env.example` for all configuration options.

**Required Environment Variables:**
- `JWT_SECRET` - 32+ character secret for JWT signing
- `DCR_AUTH_TOKEN` - 32+ character token for client registration
- `SERVICENOW_INSTANCE` - Your ServiceNow instance URL

## Security

Generate secure secrets using:
```bash
node -e "console.log(require('crypto').randomBytes(32).toString('base64url'))"
```

**Never commit:**
- `.env` file with real secrets
- `registered_clients.json` with OAuth client credentials
~~~

```
## Development

- Start development server: `npm run dev`
- Run tests: `npm test`
- Check logs: `pm2 logs` (if using PM2)

## Production Deployment

- Use cloud secret managers (not .env files)
- Configure proper HTTPS/TLS
- Enable rate limiting
- Set up monitoring and alerting

## ServiceNow Integration

- Protocol Version: 2025-03-26
- Authentication: OAuth 2.1 with PKCE
- MCP Endpoint: `https://your-domain.com/mcp`

## License

[Your chosen license]
```

Repository Structure Best Practices

Organize your repository for clarity and maintainability:

```
mcp-server/
├── .env.example           # Configuration template
├── .gitignore             # Git exclusions
├── README.md              # Project documentation
├── package.json           # Node.js dependencies
├── server.js              # Main server file
├── config/                # Configuration modules
│   ├── constants.js       # Server constants
│   └── validation.js      # Config validation
├── middleware/            # Express middleware
│   ├── auth.js            # Authentication
│   ├── cors.js            # CORS configuration
│   └── logging.js         # Request logging
├── routes/                # Endpoint handlers
│   ├── oauth.js           # OAuth 2.1 endpoints
│   └── mcp.js             # MCP protocol endpoints
├── tools/                 # MCP tool implementations
│   ├── llm-generate.js    # LLM generation tool
│   └── file-operations.js  # File tools
├── utils/                 # Utility functions
│   ├── jwt.js             # JWT token functions
│   └── pkce.js            # PKCE validation
└── tests/                 # Test files
    ├── config.test.js
    ├── oauth.test.js
    └── mcp.test.js
```

Implementation Approach Note:

The repository structure shown above reflects a modular, team-oriented organization suitable for larger projects with multiple developers. The reference implementations provided with this guide (`server-compliant.js`, `gcp-compliant.ts`) use single-file architecture for simplicity and clarity.

Both approaches are valid:

- **Single-file**: Easier to understand, faster to prototype, suitable for small teams or POC
- **Modular**: Better separation of concerns, easier to test, scales better for large teams

Choose based on your project size, team size, and long-term maintenance requirements.

Security Considerations

Pre-Commit Hooks: Consider using tools like `husky` with `lint-staged` to prevent committing secrets:

```json
{
  "husky": {
    "hooks": {
      "pre-commit": "lint-staged"
    }
  },
  "lint-staged": {
    "*.js": [
      "eslint --fix",
      "git add"
    ],
    ".env": [
      "echo 'ERROR: Cannot commit .env file' && exit 1"
    ]
  }
}
```

Secret Scanning: Enable secret scanning in GitHub/GitLab to detect accidentally committed secrets.

Branch Protection: For team projects, consider:

- Requiring pull request reviews
- Running automated tests before merge
- Protecting main/production branches

MCP and OAuth Specific Security:

Additional security considerations for MCP server implementations:

Token and Secret Security:

- Never log full JWT tokens - log only first 10-15 characters for debugging
- Rotate JWT_SECRET periodically in production (e.g., quarterly with proper token migration)
- Rotate DCR_AUTH_TOKEN periodically to prevent unauthorized client registrations
- Store all secrets in secure secret managers (not .env files) in production

Authentication Monitoring:

- Monitor failed authentication attempts (potential brute force attacks)
- Set up alerts for unusual authentication patterns (geographic anomalies, high failure rates)
- Log all OAuth client registrations for audit trail
- Track token revocation events

Rate Limiting and Abuse Prevention:

- Set up alerts when rate limits are frequently hit
- Monitor for token farming attempts (rapid DCR registrations)
- Track OAuth endpoint usage patterns
- Consider IP whitelisting for known ServiceNow IP ranges

Tool Security:

- Validate and sanitize ALL tool input parameters
- Implement input size limits for tool arguments
- Prevent directory traversal in file operations (validate paths)
- Restrict tool access to sensitive resources
- Never expose internal system details in tool error messages
- Consider tool-level rate limiting for expensive operations

Audit and Compliance:

- Retain audit logs for compliance requirements (check your industry regulations)
- Log OAuth grant events (who got tokens, when, for what scope)
- Log tool executions (which client, which tool, timestamp)
- Implement log rotation to prevent disk space issues
- Consider centralized logging (CloudWatch, Stackdriver, etc.)

Documentation Maintenance

Keep documentation synchronized with code:

When to Update README:

- Adding new MCP tools
- Changing configuration requirements
- Updating deployment procedures

- Modifying ServiceNow integration details

When to Update .env.example:

- Adding new environment variables
- Changing default values
- Deprecating old configuration options

Note: Repository practices are optional and depend on your development workflow. The core MCP server implementation works identically whether version controlled or not.

Appendix B: Alternative Storage Implementations

This appendix provides implementation patterns for different storage solutions beyond the Redis and file-based examples shown in the main documentation.

PostgreSQL Client Storage

Use Case: Centralized relational database for client credentials

Schema:

```sql
CREATE TABLE oauth_clients (
  client_id VARCHAR(255) PRIMARY KEY,
  client_secret VARCHAR(255) NOT NULL,
  client_name VARCHAR(255) NOT NULL,
  redirect_uris JSONB NOT NULL,
  grant_types JSONB NOT NULL,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX idx_client_name ON oauth_clients(client_name);
```

Implementation Pattern:

```javascript
```

```
const { Pool } = require('pg');
const pool = new Pool({
  connectionString: process.env.DATABASE_URL
});

async function storeClient(clientData) {
  await pool.query(
    'INSERT INTO oauth_clients (client_id, client_secret, client_name, redirect_uris, grant_types) VALUES ($1, $2, $3, $4, $5
    [clientData.clientId, clientData.clientSecret, clientData.clientName,
     JSON.stringify(clientData.redirectUris), JSON.stringify(clientData.grantTypes)]
  );
}

async function getClient(clientId) {
  const result = await pool.query(
    'SELECT * FROM oauth_clients WHERE client_id = $1',
    [clientId]
  );
  return result.rows[0];
}
```

MongoDB Client Storage

Use Case: Document-oriented storage for flexible client schemas

Implementation Pattern:

```
javascript
```

```
const { MongoClient } = require('mongodb');
const client = new MongoClient(process.env.MONGODB_URI);

await client.connect();
const db = client.db('mcp_server');
const clientsCollection = db.collection('oauth_clients');

async function storeClient(clientData) {
  await clientsCollection.insertOne({
    _id: clientData.clientId,
    clientSecret: clientData.clientSecret,
    clientName: clientData.clientName,
    redirectUris: clientData.redirectUris,
    grantTypes: clientData.grantTypes,
    createdAt: new Date()
  });
}

async function getClient(clientId) {
  return await clientsCollection.findOne({ _id: clientId });
}
```

File-Based Token Blacklist

Use Case: Simple persistent storage without external dependencies

Implementation Pattern:

```
javascript
```

```
const fs = require('fs');
const BLACKLIST_FILE = './revoked_tokens.json';

// Load blacklist on startup
let revokedTokens = new Map();

function loadBlacklist() {
  if (fs.existsSync(BLACKLIST_FILE)) {
    const data = JSON.parse(fs.readFileSync(BLACKLIST_FILE, 'utf8'));
    revokedTokens = new Map(Object.entries(data));
  }
}

function saveBlacklist() {
  fs.writeFileSync(BLACKLIST_FILE,
    JSON.stringify(Object.fromEntries(revokedTokens), null, 2));
}

async function revokeToken(jti, expiresIn) {
  const expirationTime = Date.now() + (expiresIn * 1000);
  revokedTokens.set(jti, expirationTime);
  saveBlacklist();
}

async function isTokenRevoked(jti) {
  const expiration = revokedTokens.get(jti);
  if (!expiration) return false;

  // Check if still within expiration window
  if (Date.now() > expiration) {
    revokedTokens.delete(jti);
    saveBlacklist();
    return false;
  }

  return true;
}

// Periodic cleanup of expired tokens
setInterval(() => {
  const now = Date.now();
  let cleaned = 0;
  for (const [jti, expiration] of revokedTokens.entries()) {
    if (now > expiration) {
      revokedTokens.delete(jti);
      cleaned++;
    }
  }
  if (cleaned > 0) {
    console.log(`Cleaned ${cleaned} expired tokens from blacklist`);
    saveBlacklist();
  }
}, 3600000); // Hourly
```

Trade-offs:

- ✅ No external dependencies
- ✅ Simple implementation
- ❌ Slower than Redis for large blacklists
- ❌ File I/O on every revocation
- ❌ Not suitable for multi-server deployments

Storage Decision Matrix

Storage Type	Best For	Pros	Cons
Redis	Production, multi-server	Fast, automatic TTL, scalable	External dependency
PostgreSQL	Enterprise, existing DB	Centralized, ACID compliance	Slower than Redis
MongoDB	Document-heavy apps	Flexible schema, TTL indexes	External dependency
File-based	POC, single-server	Simple, no dependencies	Not scalable, file I/O overhead
In-memory	Development only	Fastest	Lost on restart

Appendix C: Production Deployment Checklist

Pre-Deployment Verification

Configuration:

☐ All required environment variables set in production environment

☐ JWT_SECRET is cryptographically secure (32+ bytes) and unique per environment

- [ ] DCR_AUTH_TOKEN is cryptographically secure (32+ bytes)
- [ ] SERVICENOW_INSTANCE URL is correct
- [ ] Token lifetimes appropriate for security requirements
- [ ] No hardcoded secrets in code

Security:

- [ ] HTTPS/TLS properly configured
- [ ] CORS configured for ServiceNow instance (if not platform-handled)
- [ ] Rate limiting enabled on all endpoints
- [ ] Body parser size limits configured
- [ ] Path sanitization implemented in file operation tools
- [ ] Global error handler prevents internal detail exposure
- [ ] Audit logging captures authentication events

Storage:

- [ ] Client registry persistence configured (survives restarts)
- [ ] Token revocation storage configured (Redis/database/file)
- [ ] Authorization code cleanup scheduled
- [ ] Database/Redis connections tested and stable
- [ ] Backup strategy for client registrations

OAuth Endpoints:

- [ ] Authorization Server Metadata endpoint accessible
- [ ] DCR endpoint secured with DCR_AUTH_TOKEN
- [ ] Authorization endpoint validates PKCE properly
- [ ] Token endpoint validates client credentials
- [ ] Token revocation endpoint returns 200 OK always
- [ ] All OAuth endpoints have rate limiting

MCP Protocol:

- [ ] Initialize endpoint accessible without authentication
- [ ] Tools/list returns valid JSON Schema for all tools
- [ ] Tools/call properly routes to tool implementations
- [ ] All tools validate input parameters
- [ ] Tool error messages are user-friendly

Testing:

- [ ] Health check endpoint responds correctly
- [ ] OAuth metadata discoverable

- [ ] DCR registration tested
- [ ] Full OAuth flow tested (authorize → token exchange)
- [ ] MCP initialize handshake tested
- [ ] Tool execution tested with ServiceNow

Monitoring & Alerting:

- [ ] Server health monitoring configured
- [ ] Failed authentication alerts configured
- [ ] Rate limit hit alerts configured
- [ ] Error rate monitoring configured
- [ ] Token revocation tracking enabled

Documentation:

- [ ] README.md up to date with deployment instructions
- [ ] .env.example includes all required variables
- [ ] API documentation generated (if applicable)
- [ ] Runbook created for operations team

Post-Deployment Validation

Within 1 Hour:

- [ ] ServiceNow successfully connects and authenticates
- [ ] Tools appear in ServiceNow AI agent configuration
- [ ] Tool execution tested from ServiceNow
- [ ] Logs confirm successful requests
- [ ] No unexpected errors in server logs

Within 24 Hours:

- [ ] Monitor for memory leaks (check server memory usage)
- [ ] Verify authorization code cleanup runs
- [ ] Check token revocation blacklist size
- [ ] Review authentication success/failure rates
- [ ] Validate rate limiting not blocking legitimate traffic

Within 1 Week:

- [ ] Review audit logs for anomalies
- [ ] Verify refresh token rotation working correctly
- [ ] Check client registry remains stable
- [ ] Monitor server performance under normal load

- [ ] Validate backup/restore procedures

Appendix D: ServiceNow Connection Configuration

Configuring ServiceNow MCP Client

Prerequisites:

- MCP server deployed and accessible via HTTPS
- OAuth endpoints tested and functional
- DCR endpoint secured with DCR_AUTH_TOKEN

Step 1: Navigate to MCP Client Configuration

In ServiceNow:

1. Navigate to **AI Platform → MCP Connections**
2. Click **New** to create a new MCP connection
3. Select connection type: **Dynamic Client Registration (DCR)**

Step 2: Configure DCR Settings

Enter the following information:

- **Connection Name**: Descriptive name (e.g., "Production MCP Server")
- **MCP Endpoint URL**: Your server's MCP endpoint
  - Format: `https://your-domain.com/mcp`
  - Must be HTTPS in production
- **DCR Registration URL**: Your server's DCR endpoint
  - Format: `https://your-domain.com/register`
- **DCR Authorization Token**: The DCR_AUTH_TOKEN from your server's .env
  - This is the secure token that protects your /register endpoint
  - Must match DCR_AUTH_TOKEN configured on your server

Step 3: Test Connection

1. Click **Test Connection** button
2. ServiceNow will:
   - Call your /register endpoint with DCR token

- Receive client_id and client_secret
- Store credentials in oauth_credential table
- Test OAuth flow (authorize → token exchange)
- Test MCP initialize handshake
- Retrieve tools list

Expected Result: "Connection successful" with list of available tools

Step 4: Verify Tool Availability

1. Navigate to AI agent configuration
2. Select your MCP connection
3. Verify all tools appear in the available tools list
4. Test tool execution with sample parameters

Common ServiceNow Connection Errors

Error: "Failed to register client"

- **Cause:** DCR_AUTH_TOKEN mismatch or DCR endpoint not accessible
- **Solution:** Verify DCR token matches, check server logs, verify HTTPS accessible

Error: "OAuth authorization failed"

- **Cause:** OAuth endpoints not accessible or PKCE validation failing
- **Solution:** Check /.well-known/oauth-authorization-server endpoint, verify PKCE implementation

Error: "Tools not found"

- **Cause:** MCP endpoint authentication failing or tools/list returning empty
- **Solution:** Verify OAuth tokens being sent correctly, check tools/list response format

Error: "CORS policy error"

- **Cause:** CORS not configured for ServiceNow instance
- **Solution:** Add CORS middleware with SERVICENOW_INSTANCE origin (if platform doesn't handle it)

Error: "Connection timeout"

- **Cause:** Server not accessible from ServiceNow, firewall blocking
- **Solution:** Verify HTTPS endpoint accessible publicly, check firewall rules

ServiceNow Integration Best Practices

Tool Naming:

- Use descriptive, action-oriented names (e.g., "search_knowledge_base" not "tool1")
- Avoid special characters or spaces in tool names
- Keep names under 50 characters for UI display

Tool Descriptions:

- Write descriptions that help AI agents understand when to use the tool
- Include input/output expectations
- Mention any prerequisites or limitations

Testing Tools from ServiceNow:

1. Create test AI agent in ServiceNow
2. Configure agent to use your MCP connection
3. Test each tool with sample parameters
4. Verify responses display correctly in ServiceNow UI
5. Check server logs confirm tool execution

Monitoring ServiceNow Integration:

- Track tool execution frequency and patterns
- Monitor for failed tool calls (check error rates)
- Review AI agent usage analytics in ServiceNow
- Correlate ServiceNow user actions with MCP server logs

Reference Implementation Files

The following reference files accompany this documentation:

Complete Implementations

server-compliant.js

- **Purpose:** Local VM deployment reference implementation
- **Language:** JavaScript (Node.js)
- **Storage:** File-based client registry, Redis token blacklist

- **Tools:** LLM generation (Ollama), file operations
- **Features:** All documentation compliance improvements applied
- **Use Case:** Local development, on-premises deployment, VM-based hosting

gcp-compliant.ts

- **Purpose:** Google Cloud deployment reference implementation
- **Language:** TypeScript
- **Storage:** Firestore client registry, in-memory token blacklist
- **Tools:** A2A agent integration, utility tools (echo, calculate, timestamp, reverse)
- **Features:** All documentation compliance improvements applied
- **Use Case:** Google Cloud Run, Cloud Functions, containerized cloud deployment

Templates

server-example.js

- **Purpose:** White-label template for customization
- **Language:** JavaScript (Node.js)
- **Content:** Complete code structure with logging placeholders
- **Features:** [MUST HAVE], [RECOMMENDED], [OPTIONAL] markers throughout
- **Use Case:** Starting point for practitioners to build custom implementations
- **Requires:** Implementation of storage functions, tool functions, and configuration

.env.example

- **Purpose:** Environment configuration template
- **Content:** All required and optional environment variables
- **Features:** Comprehensive comments explaining each variable
- **Use Case:** Copy to .env and customize for your deployment

Visual Resources

oauth-flow-diagram.mermaid

- **Purpose:** OAuth 2.1 sequence diagram
- **Format:** Mermaid (renders in GitHub, GitLab, documentation sites)
- **Content:** Complete OAuth flow from DCR through token refresh
- **Use Case:** Documentation, presentations, team training

oauth-flow-diagram.svg

- **Purpose:** OAuth 2.1 sequence diagram
- **Format:** SVG (scalable vector graphics)
- **Content:** Same as Mermaid version
- **Use Case:** Embedding in PDFs, presentations, universal compatibility

How to Use Reference Files

For Local/VM Deployment:

1. Use `server-compliant.js` as your starting point
2. Copy to your project directory
3. Customize tool implementations for your use case
4. Configure .env based on .env.example
5. Deploy with PM2 or systemd

For Cloud Deployment:

1. Use `gcp-compliant.ts` as reference for cloud patterns
2. Adapt storage layer to your cloud provider (Firestore, DynamoDB, etc.)
3. Configure environment variables in cloud platform
4. Deploy as container or serverless function

For Custom Implementation:

1. Use `server-example.js` as structural template
2. Implement placeholder functions for your requirements
3. Choose storage solutions appropriate for your architecture
4. Add custom tools for your specific use case

Appendix B: Alternative Storage Implementations

This appendix provides implementation patterns for different storage solutions beyond the Redis and file-based examples shown in the main documentation.

PostgreSQL Client Storage

Use Case: Centralized relational database for client credentials in enterprise environments

Schema:

sql

```sql
CREATE TABLE oauth_clients (
  client_id VARCHAR(255) PRIMARY KEY,
  client_secret VARCHAR(255) NOT NULL,
  client_name VARCHAR(255) NOT NULL,
  redirect_uris JSONB NOT NULL,
  grant_types JSONB NOT NULL,
  use_pkce BOOLEAN DEFAULT true,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX idx_client_name ON oauth_clients(client_name);
```

Implementation Pattern (JavaScript):

javascript

```
const { Pool } = require('pg');

const pool = new Pool({
  connectionString: process.env.DATABASE_URL,
  max: 20,  // Connection pool size
  idleTimeoutMillis: 30000,
  connectionTimeoutMillis: 2000
});

async function storeClient(clientData) {
  await pool.query(
    `INSERT INTO oauth_clients
     (client_id, client_secret, client_name, redirect_uris, grant_types, use_pkce)
     VALUES ($1, $2, $3, $4, $5, $6)`,
    [
      clientData.clientId,
      clientData.clientSecret,
      clientData.clientName,
      JSON.stringify(clientData.redirectUris),
      JSON.stringify(clientData.grantTypes),
      clientData.usePkce
    ]
  );
  console.log(`[POSTGRES] Client stored: ${clientData.clientId}`);
}

async function getClient(clientId) {
  const result = await pool.query(
    'SELECT * FROM oauth_clients WHERE client_id = $1',
    [clientId]
  );

  if (result.rows.length === 0) return null;

  const row = result.rows[0];
  return {
    clientId: row.client_id,
    clientSecret: row.client_secret,
    clientName: row.client_name,
    redirectUris: JSON.parse(row.redirect_uris),
    grantTypes: JSON.parse(row.grant_types),
    usePkce: row.use_pkce,
    createdAt: row.created_at
  };
}

async function loadAllClients() {
  const result = await pool.query('SELECT * FROM oauth_clients');
  const clients = new Map();

  result.rows.forEach(row => {
    clients.set(row.client_id, {
      clientId: row.client_id,
      clientSecret: row.client_secret,
      clientName: row.client_name,
      redirectUris: JSON.parse(row.redirect_uris),
      grantTypes: JSON.parse(row.grant_types),
      usePkce: row.use_pkce
    });
  });

  console.log(`[POSTGRES] Loaded ${clients.size} client(s)`);
  return clients;
}
```

MongoDB Client Storage

Use Case: Document-oriented storage for flexible schemas and cloud-native applications

Implementation Pattern (JavaScript):

```
javascript
```

```
const { MongoClient } = require('mongodb');

const client = new MongoClient(process.env.MONGODB_URI);
await client.connect();

const db = client.db('mcp_server');
const clientsCollection = db.collection('oauth_clients');

// Create index for faster lookups
await clientsCollection.createIndex({ client_id: 1 }, { unique: true });

async function storeClient(clientData) {
  await clientsCollection.insertOne({
    _id: clientData.clientId,
    clientSecret: clientData.clientSecret,
    clientName: clientData.clientName,
    redirectUris: clientData.redirectUris,
    grantTypes: clientData.grantTypes,
    usePkce: clientData.usePkce,
    createdAt: new Date()
  });
  console.log(`[MONGODB] Client stored: ${clientData.clientId}`);
}

async function getClient(clientId) {
  const doc = await clientsCollection.findOne({ _id: clientId });
  if (!doc) return null;

  return {
    clientId: doc._id,
    clientSecret: doc.clientSecret,
    clientName: doc.clientName,
    redirectUris: doc.redirectUris,
    grantTypes: doc.grantTypes,
    usePkce: doc.usePkce,
    createdAt: doc.createdAt
  };
}

async function loadAllClients() {
  const cursor = clientsCollection.find({});
  const clients = new Map();
  await cursor.forEach(doc => {
    clients.set(doc._id, {
      clientId: doc._id,
      clientSecret: doc.clientSecret,
      clientName: doc.clientName,
      redirectUris: doc.redirectUris,
      grantTypes: doc.grantTypes,
      usePkce: doc.usePkce
    });
  });

  console.log(`[MONGODB] Loaded ${clients.size} client(s)`);
  return clients;
}
```

File-Based Token Blacklist

Use Case: Simple persistent token revocation without external dependencies

Implementation Pattern (JavaScript):

javascript

```
const fs = require('fs');
const path = require('path');

const BLACKLIST_FILE = path.join(__dirname, 'revoked_tokens.json');

// In-memory cache with file persistence
let revokedTokens = new Map();

function loadBlacklist() {
  try {
    if (fs.existsSync(BLACKLIST_FILE)) {
      const data = JSON.parse(fs.readFileSync(BLACKLIST_FILE, 'utf8'));

      // Load as Map with expiration times
      for (const [jti, expiration] of Object.entries(data)) {
        // Only load tokens that haven't expired yet
        if (expiration > Date.now()) {
          revokedTokens.set(jti, expiration);
        }
      }

      console.log(`[BLACKLIST] Loaded ${revokedTokens.size} revoked token(s) from file`);
    }
  } catch (error) {
    console.error('[BLACKLIST] Failed to load:', error.message);
  }
}

function saveBlacklist() {
  try {
    const data = Object.fromEntries(revokedTokens);
    fs.writeFileSync(BLACKLIST_FILE, JSON.stringify(data, null, 2), 'utf8');
    console.log(`[BLACKLIST] Saved ${revokedTokens.size} revoked token(s) to file`);
  } catch (error) {
    console.error('[BLACKLIST] Failed to save:', error.message);
  }
}

async function revokeToken(jti, expiresInSeconds) {
  const expirationTime = Date.now() + (expiresInSeconds * 1000);
  revokedTokens.set(jti, expirationTime);
  saveBlacklist();
  console.log(`[BLACKLIST] Token revoked: ${jti.substring(0, 10)}...`);
}

async function isTokenRevoked(jti) {
  const expiration = revokedTokens.get(jti);

  if (!expiration) {
    return false;  // Not in blacklist
  }

  // Check if token expiration has passed (natural expiry)
  if (Date.now() > expiration) {
    // Remove from blacklist (cleanup)
    revokedTokens.delete(jti);
    saveBlacklist();
    return false;
  }

  return true;  // Still revoked
}

// Periodic cleanup of naturally expired tokens from blacklist
setInterval(() => {
  const now = Date.now();
  let cleaned = 0;

  for (const [jti, expiration] of revokedTokens.entries()) {
    if (now > expiration) {
      revokedTokens.delete(jti);
      cleaned++;
    }
  }

  if (cleaned > 0) {
    console.log(`[BLACKLIST] Cleaned ${cleaned} expired token(s) from blacklist`);
    saveBlacklist();
  }
}, 3600000); // Run every hour

// Load blacklist on server startup
loadBlacklist();
```

Trade-offs:

- ✅ No external dependencies (Redis, database)

- ✅ Simple implementation
- ✅ Survives server restarts
- ❌ File I/O on every revocation (slower than Redis)
- ❌ Not suitable for high-concurrency or multi-server deployments
- ❌ Larger blacklists = slower file writes

Storage Decision Matrix

Storage Type	Best For	Pros	Cons
Redis	Production, multi-server, high-traffic	Fast (in-memory), automatic TTL, horizontal scaling	External dependency, requires Redis server
PostgreSQL	Enterprise, existing DB infrastructure	Centralized, ACID compliance, powerful queries	Slower than Redis, more complex setup
MongoDB	Document-heavy apps, cloud-native	Flexible schema, TTL indexes, cloud-managed options	External dependency, eventual consistency
Firestore	Google Cloud deployments	Managed service, real-time sync, auto-scaling	GCP-specific, cost at scale
File-based	POC, single-server, low-traffic	Simple, no dependencies, easy debugging	File I/O overhead, not scalable, single-server only
In-memory	Development/testing only	Fastest possible	Lost on restart, not production-suitable

Recommendation: Choose storage based on deployment architecture:

- **Single-server deployment:** File-based or Redis
- **Multi-server deployment:** Redis, PostgreSQL, or MongoDB
- **Cloud-native:** Platform-specific (Firestore for GCP, DynamoDB for AWS)
- **Enterprise:** PostgreSQL or MongoDB with existing database infrastructure

---

Appendix C: Production Deployment Checklist

Pre-Deployment Verification

Configuration Validation:

- [ ] All required environment variables set in production environment
- [ ] JWT_SECRET is cryptographically secure (32+ bytes) and environment-specific
- [ ] DCR_AUTH_TOKEN is cryptographically secure (32+ bytes)
- [ ] SERVICENOW_INSTANCE URL is correct (https://instance.service-now.com)
- [ ] Token lifetimes appropriate for your security requirements
- [ ] No hardcoded secrets anywhere in code
- [ ] Configuration validation function passes all checks

Security Hardening:

- [ ] HTTPS/TLS properly configured (valid certificate, strong ciphers)
- [ ] CORS configured for ServiceNow instance (or verified platform handles it)
- [ ] Rate limiting enabled on all endpoints (OAuth and MCP)
- [ ] Body parser size limits configured (1MB JSON, 100KB URL-encoded)
- [ ] Path sanitization implemented in all file operation tools
- [ ] Global error handler prevents internal implementation detail exposure
- [ ] Audit logging captures authentication and tool execution events
- [ ] No debug/verbose logging in production (sensitive data exposure)

Storage and Persistence:

- [ ] Client registry persistence configured (survives server restarts)
- [ ] Token revocation storage configured and tested (Redis/database/file)
- [ ] Authorization code cleanup scheduled (prevents memory leaks)
- [ ] Database/Redis connections tested and stable
- [ ] Backup strategy implemented for client registrations
- [ ] Storage credentials secured (not in code, use secret managers)

OAuth 2.1 Endpoints:

- [ ] Authorization Server Metadata endpoint accessible (/.well-known/oauth-authorization-server)
- [ ] DCR endpoint secured with DCR_AUTH_TOKEN
- [ ] Authorization endpoint validates PKCE parameters (code_challenge, method=S256)
- [ ] Token endpoint validates client credentials correctly
- [ ] Token endpoint performs PKCE validation
- [ ] Token revocation endpoint always returns 200 OK (RFC 7009 compliance)
- [ ] All OAuth endpoints have rate limiting configured

MCP Protocol:

- ☐ Initialize endpoint accessible without authentication
- ☐ Initialize returns correct protocol version (2025-03-26 or 2025-06-18)
- ☐ Tools/list returns valid JSON Schema for all tools
- ☐ Tools/call properly routes to tool implementations
- ☐ All tools validate required input parameters
- ☐ Tool error messages are user-friendly (no stack traces to clients)
- ☐ Notifications handler acknowledges without response body

Testing:

- ☐ Health check endpoint responds correctly (GET /health)
- ☐ OAuth metadata endpoints accessible and return valid JSON
- ☐ DCR registration tested with valid DCR token
- ☐ Full OAuth flow tested (authorize → token exchange → MCP request)
- ☐ MCP initialize handshake tested
- ☐ All tools tested with valid and invalid parameters
- ☐ Token refresh flow tested
- ☐ Token revocation tested

Monitoring and Observability:

- ☐ Server health monitoring configured (uptime, memory, CPU)
- ☐ Failed authentication alerts configured
- ☐ Rate limit threshold alerts configured
- ☐ Error rate monitoring configured (5xx responses)
- ☐ Token revocation tracking enabled
- ☐ Tool execution metrics captured
- ☐ Log aggregation configured (CloudWatch, Stackdriver, ELK, etc.)

Documentation:

- ☐ README.md updated with deployment instructions
- ☐ .env.example includes all required and optional variables
- ☐ Runbook created for operations team (restart procedures, troubleshooting)
- ☐ Architecture diagram created (optional but helpful)

Post-Deployment Validation

Within 1 Hour of Deployment:

- ☐ Server starts without errors

- [ ] Health endpoint returns 200 OK
- [ ] OAuth metadata endpoints accessible
- [ ] ServiceNow test connection successful
- [ ] ServiceNow authenticates via OAuth successfully
- [ ] Tools appear in ServiceNow AI agent configuration
- [ ] At least one tool execution tested from ServiceNow
- [ ] Server logs confirm successful requests (no errors)
- [ ] No unexpected errors in application logs

Within 24 Hours:

- [ ] Monitor for memory leaks (check memory usage trends)
- [ ] Verify authorization code cleanup executes (check cleanup logs)
- [ ] Check token revocation blacklist size (should grow slowly)
- [ ] Review authentication success vs failure rates (failures should be minimal)
- [ ] Validate rate limiting not blocking legitimate ServiceNow traffic
- [ ] Verify no CORS errors in ServiceNow
- [ ] Check client registry file/database integrity

Within 1 Week:

- [ ] Review comprehensive audit logs for anomalies
- [ ] Verify refresh token rotation working correctly (check rotation_count in logs)
- [ ] Validate client registry remains stable (no corruption)
- [ ] Monitor server performance under normal ServiceNow usage
- [ ] Test and validate backup/restore procedures
- [ ] Review error logs and address any recurring issues
- [ ] Validate monitoring alerts are working (trigger test alerts)

Rollback Plan

Prepare rollback procedures before deploying:

Rollback Steps:

1. Keep previous version available (containerized or backed up)
2. Document configuration differences between versions
3. Test rollback procedure in staging environment
4. Have database migration rollback scripts ready (if schema changed)
5. Communicate rollback window to ServiceNow users

When to Rollback:

- ServiceNow connection failures exceeding 5% of requests
- Tool execution failures exceeding 10% of calls
- Server crashes or memory issues
- Authentication failures preventing ServiceNow access
- Critical security vulnerability discovered

Appendix D: ServiceNow Connection Configuration

Configuring ServiceNow MCP Client

Prerequisites:

- MCP server deployed and accessible via HTTPS URL
- OAuth endpoints tested and functional
- DCR endpoint secured with DCR_AUTH_TOKEN
- At least one tool implemented and tested

Step-by-Step Configuration

Step 1: Navigate to MCP Client Configuration in ServiceNow

1. Log in to your ServiceNow instance as administrator
2. Navigate to **AI Platform → MCP Connections** (or search "MCP Connections" in filter navigator)
3. Click **New** to create a new MCP connection
4. Select connection type: **OAuth 2.1 with Dynamic Client Registration**

Step 2: Enter Connection Details

Fill in the connection form:

Basic Information:

- **Connection Name**: Descriptive identifier (e.g., "Production MCP Server - AI Tools")
- **Description**: Optional description of what this server provides

MCP Endpoint Configuration:

- **MCP Endpoint URL**: Your server's MCP endpoint
  - Format: `https://your-domain.com/mcp`

    - Must be HTTPS in production (required by ServiceNow)
    - Must be publicly accessible from ServiceNow cloud

Dynamic Client Registration (DCR):

- **DCR Registration URL**: Your server's DCR endpoint
    - Format: `https://your-domain.com/register`
- **DCR Authorization Token**: The DCR_AUTH_TOKEN from your server's environment
    - Copy the exact value from your server's .env file
    - This is the Bearer token that protects your /register endpoint
    - ServiceNow will send this when registering as OAuth client

Step 3: Save and Test Connection

1. Click **Save** (ServiceNow validates form inputs)
2. Click **Test Connection** button
3. ServiceNow will perform these steps:
    - Call DCR endpoint with authorization token
    - Receive client_id and client_secret
    - Store OAuth credentials in oauth_credential table
    - Initiate OAuth authorization flow
    - Exchange authorization code for tokens
    - Call MCP initialize endpoint
    - Retrieve tools list via tools/list

Expected Result:

- ✅ "Connection successful" message
- ✅ OAuth client credentials generated and stored
- ✅ List of available tools displayed
- ✅ Connection status shows "Active"

Step 4: Verify Tools in AI Agent

1. Navigate to **AI Platform** → **AI Agents** (or create new AI agent)
2. Edit an existing agent or create new one
3. In agent configuration, select **Tools** tab

4. Your MCP connection should appear in available connections
5. Expand your connection to see available tools
6. Enable tools you want the AI agent to use

Troubleshooting ServiceNow Connection

Error: "Failed to register client - Unauthorized"

- **Cause:** DCR_AUTH_TOKEN mismatch between ServiceNow and server
- **Solution:**
  - Verify DCR token in ServiceNow matches server .env exactly
  - Check server logs for DCR endpoint 401 errors
  - Ensure DCR_AUTH_TOKEN is at least 32 characters

Error: "OAuth authorization failed"

- **Cause:** OAuth endpoints not accessible or PKCE validation failing
- **Solution:**
  - Test /.well-known/oauth-authorization-server endpoint directly
  - Verify /oauth/authorize endpoint accessible
  - Check server logs for PKCE validation errors
  - Ensure server validates S256 challenge method

Error: "Tools list empty" or "No tools found"

- **Cause:** Authentication failing or tools/list returning empty array
- **Solution:**
  - Verify OAuth tokens being sent in Authorization header
  - Check tools/list handler returns proper JSON-RPC format
  - Verify tools array is not empty in your code
  - Check server logs for tools/list request errors

Error: "CORS policy: No 'Access-Control-Allow-Origin' header"

- **Cause:** CORS not configured for ServiceNow instance
- **Solution:**
  - Add CORS middleware with ServiceNow instance as origin
  - Verify SERVICENOW_INSTANCE in .env matches exactly

    - For Cloud platforms: Check platform CORS settings
    - Test with: `curl -H "Origin: https://instance.service-now.com" https://your-domain.com/health`

Error: "Connection timeout" or "Failed to connect"

- **Cause:** Server not accessible from ServiceNow cloud, firewall blocking
- **Solution:**
    - Verify HTTPS endpoint accessible from external network
    - Test URL in browser: https://your-domain.com/health
    - Check firewall rules allow incoming HTTPS (port 443)
    - Verify DNS resolves correctly
    - For VPN/private deployments: Ensure ServiceNow can route to server

Error: "Invalid protocol version"

- **Cause:** Protocol version mismatch between server and ServiceNow
- **Solution:**
    - Check initialize handler returns protocolVersion: "2025-03-26" or "2025-06-18"
    - Verify ServiceNow version supports your protocol version
    - Update server to match ServiceNow-supported version

ServiceNow Integration Best Practices

Tool Naming Conventions:

- Use descriptive, action-oriented names: `search_knowledge_base`, `create_ticket`, `query_database`
- Avoid generic names: `tool1`, `function`, `api_call`
- Keep names under 50 characters for UI display
- Use snake_case for consistency with MCP conventions
- No special characters beyond underscore

Tool Descriptions for AI Agents:

- Write descriptions that help AI determine when to use the tool
- Include what the tool does, what inputs it needs, what it returns
- Mention any prerequisites or limitations
- Example: "Search the company knowledge base for articles matching the query. Returns top 5 relevant articles with titles and summaries. Requires query string of 3+ characters."

Input Schema Best Practices:

- Use descriptive parameter names and descriptions
- Set appropriate constraints (minLength, maxLength, minimum, maximum)
- Mark truly required fields as "required"
- Provide defaults for optional parameters
- Use enums for parameters with fixed choices

Testing Tools from ServiceNow:

1. Create test AI agent in ServiceNow
2. Configure agent to use your MCP connection
3. Test each tool individually with known-good parameters
4. Test error cases (missing parameters, invalid values)
5. Verify tool responses display correctly in ServiceNow UI
6. Check server logs confirm tool execution and parameters

Monitoring ServiceNow Integration:

- Track tool execution frequency (which tools are most used)
- Monitor tool execution latency (response times)
- Review failed tool calls (error patterns)
- Correlate ServiceNow AI agent usage with MCP server metrics
- Set up alerts for tool execution failures

Managing Tool Changes:

- Adding new tools: Update tools array, test locally, re-test connection in ServiceNow
- Modifying existing tools: Consider backward compatibility if AI agents already use them
- Removing tools: Verify no ServiceNow AI agents actively using the tool
- Changing tool schemas: Update descriptions, test with ServiceNow

Document Status

- **Part:** 5 of 5
- **Version:** 1.0
- **Last Updated:** January 29, 2026
- **Status:** Complete