

MCP Architecture Patterns for Solution Architects

This document outlines architectural patterns and design considerations for building enterprise-grade MCP servers.

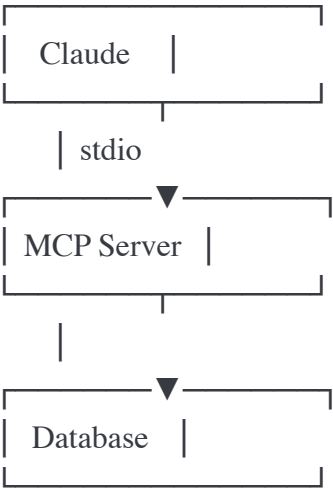
Table of Contents

- 1. [System Architecture Patterns](#)
- 2. [Integration Patterns](#)
- 3. [Data Flow Patterns](#)
- 4. [Security Patterns](#)
- 5. [Scalability Patterns](#)
- 6. [Real-World Reference Architectures](#)

System Architecture Patterns

Pattern 1: Simple Direct Integration

When to use: Single data source, low complexity, proof of concept



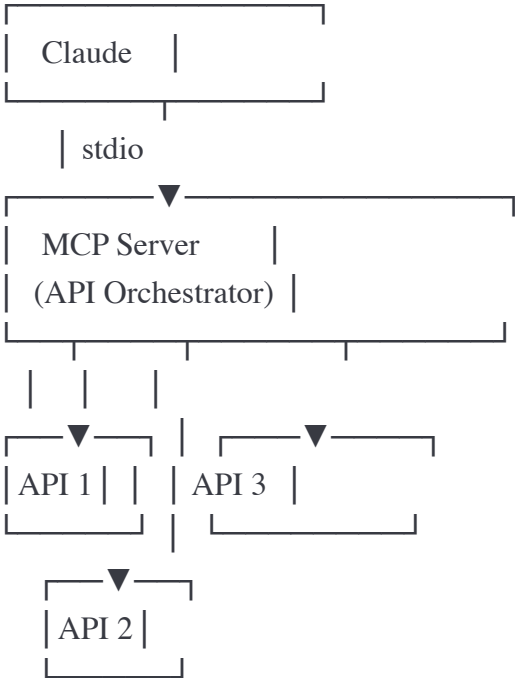
Characteristics:

- Single MCP server
- Direct database connection
- Minimal latency
- Easy to develop and debug
- Limited scalability

Example Use Case: Personal productivity tool accessing local SQLite database

Pattern 2: API Gateway Integration

When to use: Multiple external APIs, need for API management, rate limiting



Characteristics:

- Central orchestration point
- Unified error handling
- Request/response transformation
- API key management
- Circuit breaker patterns

Example Use Case: CRM integration aggregating data from Salesforce, HubSpot, and Zendesk

Implementation Tips:



python

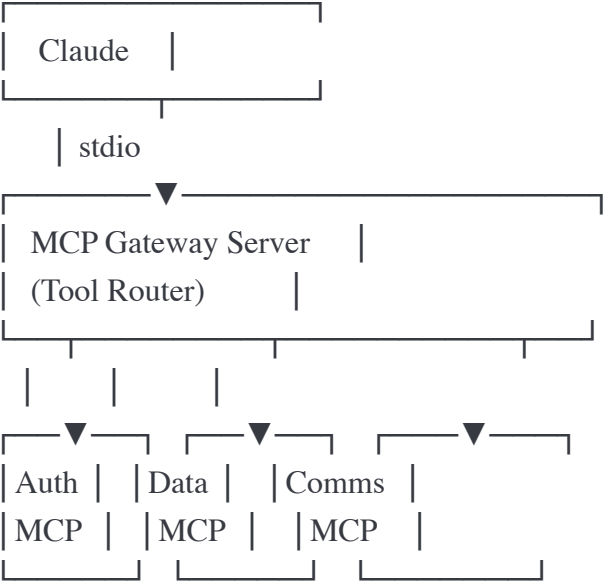
```
class APIOrchestrator:
    def __init__(self):
        self.clients = {
            'salesforce': SalesforceClient(),
            'hubspot': HubSpotClient(),
            'zendesk': ZendeskClient()
        }

    async def get_customer_360(self, customer_id: str):
        # Parallel API calls
        results = await asyncio.gather(
            self.clients['salesforce'].get_account(customer_id),
            self.clients['hubspot'].get_contact(customer_id),
            self.clients['zendesk'].get_tickets(customer_id),
            return_exceptions=True
        )

        # Merge and return unified view
        return merge_customer_data(results)
```

Pattern 3: Microservices Architecture

When to use: Multiple domains, team autonomy, need for independent scaling



Characteristics:

- Domain-separated MCP servers
- Independent deployment
- Technology diversity
- Fault isolation
- Complex orchestration

Example Use Case: Enterprise platform with separate auth, data, and communication domains

Implementation Pattern:



python

```
# Gateway server routes to appropriate domain server
class MCPGateway:
    def __init__(self):
        self.servers = {
            'auth': AuthMCPClient(),
            'data': DataMCPClient(),
            'comms': CommsMCPClient()
        }

    async def call_tool(self, name: str, args: dict):
        # Route based on tool name prefix
        domain = name.split('_')[0] # e.g., "auth_login" → "auth"

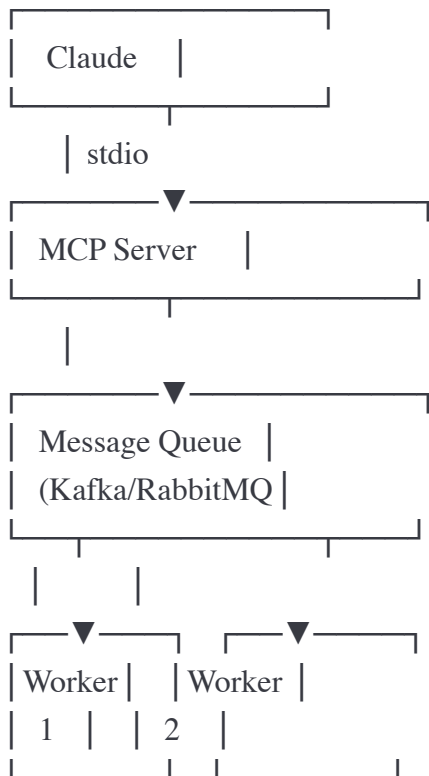
        if domain in self.servers:
            return await self.servers[domain].call_tool(name, args)

        raise ValueError(f"Unknown domain: {domain}")
```

Pattern 4: Event-Driven Architecture

When to use: Async operations, real-time updates, complex workflows





Characteristics:

- Asynchronous processing
- Long-running operations
- Scalable workers
- Guaranteed delivery
- Complex state management

Example Use Case: Data processing pipeline with ETL operations

Integration Patterns

Pattern 5: Database Integration

Multi-Database Access:



python

```
class DatabaseMCPServer:
```

```
    def __init__(self):
```

```
        self.postgres = PostgresPool()
```

```
        self.mongo = MongoClient()
```

```
        self.redis = RedisClient()
```

```
    async def unified_query(self, entity: str, id: str):
```

```
        """Query across multiple databases"""
```

```
        # Get relational data
```

```
        sql_data = await self.postgres.fetch(
```

```
            "SELECT * FROM users WHERE id = $1", id
```

```
        )
```

```
        # Get document data
```

```
        doc_data = await self.mongo.find_one(
```

```
            "user_profiles", {"user_id": id}
```

```
        )
```

```
        # Get cached data
```

```
        cache_data = await self.redis.get(f"user:{id}")
```

```
        return merge_data(sql_data, doc_data, cache_data)
```

Best Practices:

- Use connection pooling
- Implement query timeouts
- Cache frequently accessed data
- Use read replicas for read-heavy workloads
- Implement retry logic with exponential backoff

Pattern 6: File System Integration

Secure File Access:



python

```

class FileSystemMCPServer:
    def __init__(self, allowed_paths: list[str]):
        self.allowed_paths = [Path(p).resolve() for p in allowed_paths]

    def is_path_allowed(self, path: Path) -> bool:
        """Prevent directory traversal attacks"""
        resolved = path.resolve()
        return any(
            resolved.is_relative_to(allowed)
            for allowed in self.allowed_paths
        )

    async def read_file(self, file_path: str) -> str:
        path = Path(file_path)

        if not self.is_path_allowed(path):
            raise SecurityError("Access denied")

        async with aiofiles.open(path, 'r') as f:
            return await f.read()

```

Security Considerations:

- Whitelist allowed directories
- Validate file paths
- Check file permissions
- Limit file size
- Scan for malware
- Audit file access

Pattern 7: External API Integration

Resilient API Client:



python

```
from tenacity import retry, stop_after_attempt, wait_exponential
```

```
class ResilientAPIClient:
```

```
    def __init__(self, base_url: str, api_key: str):
        self.base_url = base_url
        self.session = aiohttp.ClientSession(
            headers={"Authorization": f"Bearer {api_key}"},
            timeout=aiohttp.ClientTimeout(total=30)
        )
        self.circuit_breaker = CircuitBreaker()

    @retry(
        stop=stop_after_attempt(3),
        wait=wait_exponential(multiplier=1, min=2, max=10)
    )
    async def request(self, method: str, endpoint: str, **kwargs):
        # Check circuit breaker
        if not self.circuit_breaker.allow_request():
            raise ServiceUnavailableError("Circuit breaker open")

        try:
            async with self.session.request(
                method,
                f"{self.base_url}/{endpoint}",
                **kwargs
            ) as response:
                response.raise_for_status()
                self.circuit_breaker.record_success()
                return await response.json()

        except aiohttp.ClientError as e:
            self.circuit_breaker.record_failure()
            raise
```

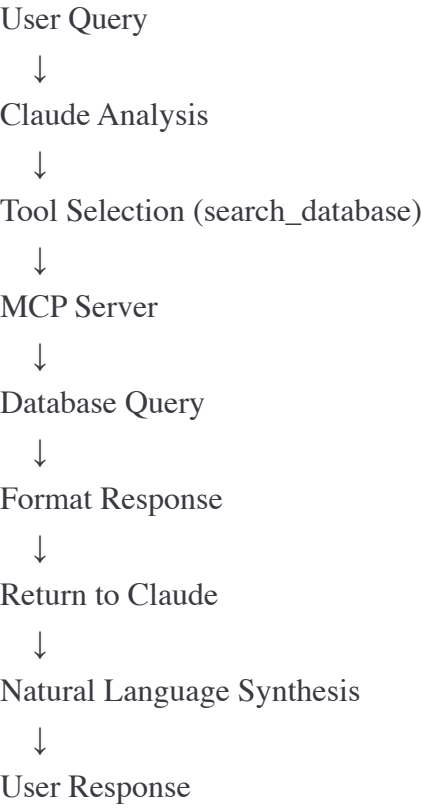
Patterns to Implement:

- Circuit breaker
 - Retry with exponential backoff
 - Timeout handling
 - Rate limiting
 - Request/response caching
 - Connection pooling
-

Data Flow Patterns

Pattern 8: Request/Response Flow

Synchronous Pattern:



Best for: Simple queries, immediate results, <5 second operations

Pattern 9: Async Workflow Pattern

Long-Running Operations:



User Request



Claude → MCP Server: start_analysis(data)



MCP Server: Returns job_id immediately



Claude → User: "Analysis started, ID: job_123"



[Background Processing]



User: "What's the status of job_123?"



Claude → MCP Server: check_status(job_123)



MCP Server: Returns current progress



Claude → User: "Analysis 75% complete"

Implementation:



python

```

class AsyncJobMCPServer:
    def __init__(self):
        self.jobs = {}
        self.queue = asyncio.Queue()

    async def start_job(self, job_type: str, params: dict) -> str:
        job_id = str(uuid.uuid4())

        self.jobs[job_id] = {
            'status': 'queued',
            'progress': 0,
            'result': None
        }

        # Add to queue for processing
        await self.queue.put((job_id, job_type, params))

        return job_id

    async def get_job_status(self, job_id: str) -> dict:
        if job_id not in self.jobs:
            raise ValueError(f"Job {job_id} not found")

        return self.jobs[job_id]

```

Best for: Data processing, reports, large computations, batch operations

Pattern 10: Streaming Pattern

Real-Time Data Streams:



python

```
async def stream_logs(self, service: str) -> AsyncIterator[str]:  
    """Stream logs in real-time"""
```

```
async with aiohttp.ClientSession() as session:  
    async with session.ws_connect(  
        f"wss://logs.example.com/stream/{service}"  
    ) as ws:  
        async for msg in ws:  
            if msg.type == aiohttp.WSMsgType.TEXT:  
                log_entry = json.loads(msg.data)  
                yield format_log_entry(log_entry)
```

Best for: Logs, metrics, real-time events, monitoring

Security Patterns

Pattern 11: Authentication & Authorization

Multi-Layer Security:



python

```
class SecureMCPServer:
```

```
    def __init__(self):
        self.auth = AuthProvider()
        self.rbac = RBACManager()

    async def call_tool(self, name: str, args: dict, context: dict):
        # 1. Authenticate user
        user = await self.auth.verify_token(context.get('token'))
        if not user:
            raise AuthenticationError()

        # 2. Check authorization
        if not self.rbac.can_execute(user, name):
            raise AuthorizationError(
                f"User {user.id} cannot execute {name}"
            )

        # 3. Audit log
        await self.audit_log(user, name, args)

        # 4. Execute with user context
        return await self.execute_tool(name, args, user)
```

Security Layers:

1. **Authentication:** Verify identity (API keys, OAuth, JWT)
 2. **Authorization:** Check permissions (RBAC, ABAC)
 3. **Audit Logging:** Track all actions
 4. **Input Validation:** Prevent injection attacks
 5. **Rate Limiting:** Prevent abuse
 6. **Data Encryption:** Protect data in transit and at rest
-

Pattern 12: Secrets Management

Using External Secrets Manager:



python

```

import boto3
from functools import lru_cache

class SecretsManager:
    def __init__(self):
        self.client = boto3.client('secretsmanager')

    @lru_cache(maxsize=100)
    def get_secret(self, secret_name: str) -> str:
        """Get secret with caching"""
        response = self.client.get_secret_value(
            SecretId=secret_name
        )
        return response['SecretString']

    def get_database_credentials(self) -> dict:
        secret = self.get_secret('prod/database')
        return json.loads(secret)

# Usage
secrets = SecretsManager()
db_config = secrets.get_database_credentials()

```

Best Practices:

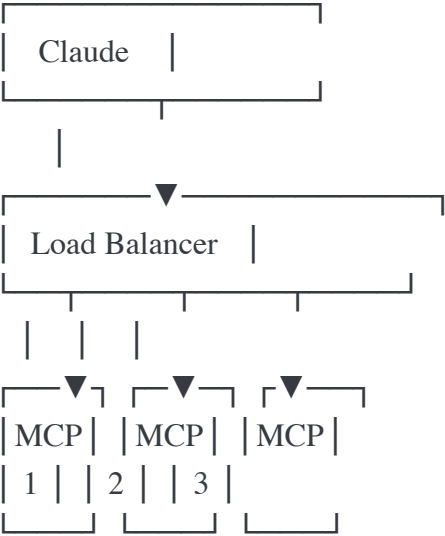
- Never hardcode secrets
- Use environment variables for local dev
- Use secrets managers for production (AWS Secrets Manager, HashiCorp Vault, etc.)
- Rotate secrets regularly
- Use IAM roles when possible
- Audit secret access

Scalability Patterns

Pattern 13: Horizontal Scaling

Load Balanced MCP Servers:





Implementation Considerations:

- Stateless servers (session data in external store)
- Shared cache (Redis, Memcached)
- Consistent hashing for data distribution
- Health checks and auto-scaling
- Connection pooling

Pattern 14: Caching Strategy

Multi-Level Caching:



python

class **CachedMCPServer**:

def **__init__**(self):

self.memory_cache = {} *# L1: In-memory*

self.redis = Redis() *# L2: Distributed*

self.db = Database() *# L3: Source of truth*

async def **get_data**(self, key: **str**):

L1: Check memory cache

if key **in** self.memory_cache:

return self.memory_cache[key]

L2: Check Redis

cached = **await** self.redis.get(key)

if cached:

self.memory_cache[key] = cached

return cached

L3: Query database

data = **await** self.db.query(key)

Update caches

await self.redis.set(key, data, ex=3600) *# 1 hour*

self.memory_cache[key] = data

return data

Caching Strategy:

- **L1 (Memory)**: Fastest, smallest, process-local
- **L2 (Redis)**: Fast, shared across servers
- **L3 (Database)**: Slowest, source of truth

Cache Invalidation:



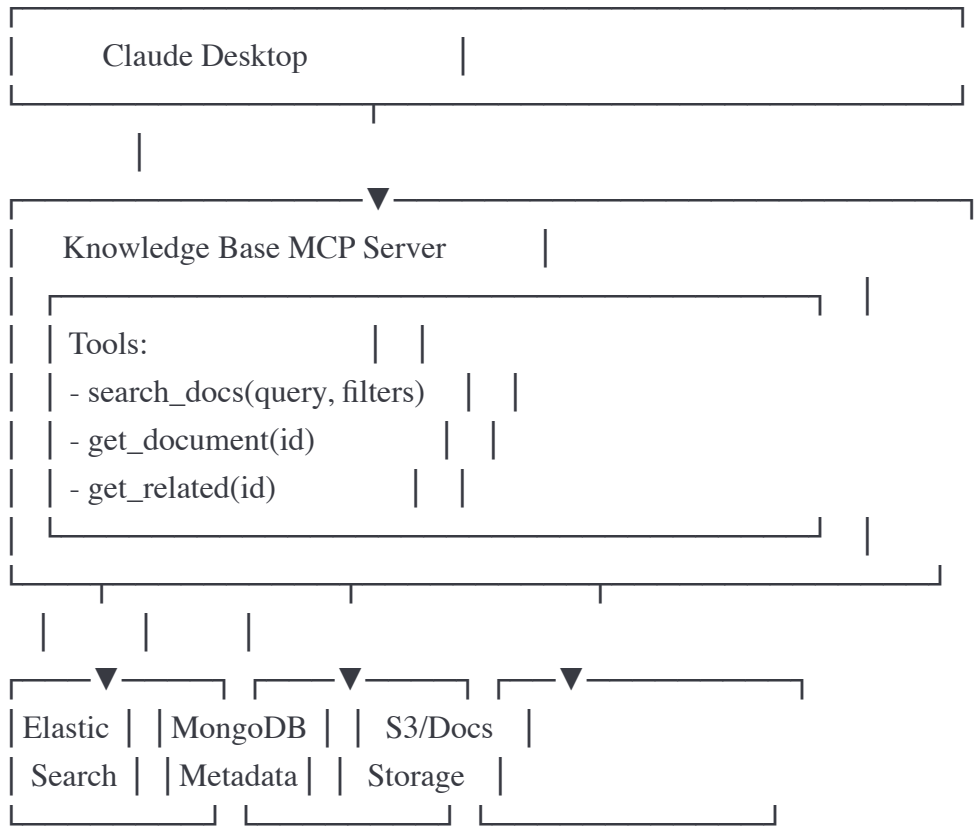
python


```
async def update_data(self, key: str, value: any):  
    # Update source  
    await self.db.update(key, value)  
  
    # Invalidate caches  
    self.memory_cache.pop(key, None)  
    await self.redis.delete(key)
```

Real-World Reference Architectures

Architecture 1: Enterprise Knowledge Base

Use Case: Company-wide documentation search



Key Features:

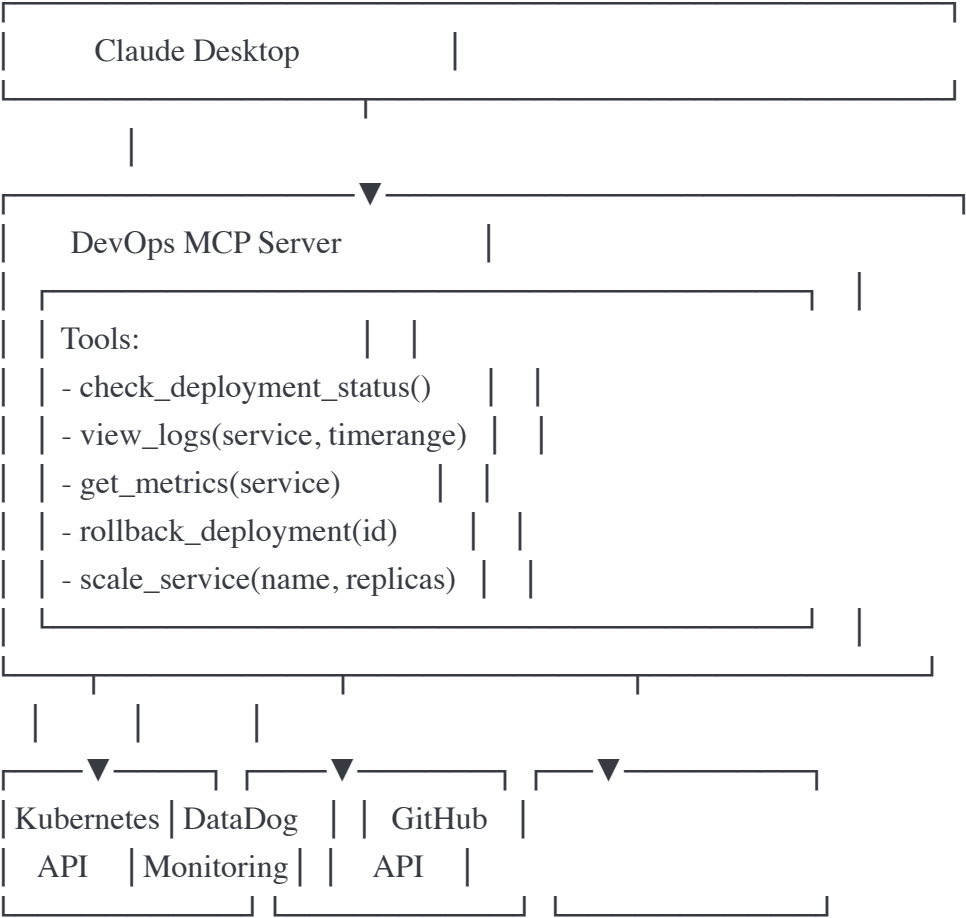
- Vector search for semantic similarity
- Metadata filtering
- Document versioning
- Access control

Tech Stack:

- MCP Server: Python + FastMCP
- Search: Elasticsearch or Pinecone
- Metadata: MongoDB
- Storage: S3 or Azure Blob
- Auth: OAuth 2.0

Architecture 2: DevOps Automation Platform

Use Case: Infrastructure management and monitoring

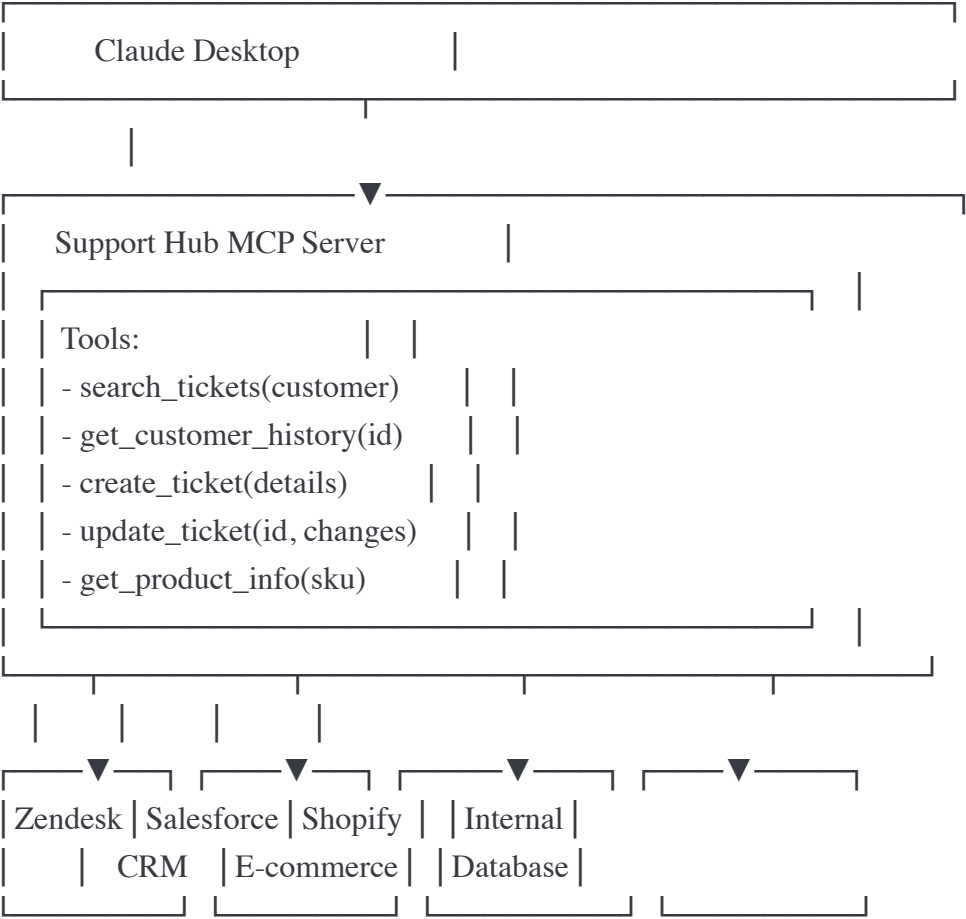


Safety Features:

- Approval workflows for destructive operations
- Audit logging of all actions
- Rate limiting
- Read-only mode by default
- Confirmation prompts

Architecture 3: Customer Support Hub

Use Case: Unified customer support interface



Data Aggregation:

- Real-time customer 360° view
- Cross-platform ticket search
- Order history integration
- Product catalog access

Decision Matrix: Choosing the Right Pattern

Requirement	Recommended Pattern
Single data source, simple queries	Direct Integration
Multiple external APIs	API Gateway
Multiple domains, team autonomy	Microservices
Long-running operations	Async Workflow
Real-time streaming	Streaming Pattern
High read volume	Caching Strategy
Strict security requirements	Multi-Layer Security
Need for independent scaling	Horizontal Scaling
Complex data transformations	Event-Driven Architecture

Performance Optimization Checklist

Connection Management

- ✔ Use connection pooling (min: 5, max: 20)
- ✔ Set appropriate timeouts (30-60s)
- ✔ Implement keepalive
- ✔ Close connections properly

Caching

- ✔ Cache expensive operations
- ✔ Use appropriate TTLs
- ✔ Implement cache warming
- ✔ Monitor cache hit rates (target: >80%)

Error Handling

- ✔ Implement circuit breakers
- ✔ Use exponential backoff
- ✔ Set maximum retry attempts (3-5)
- ✔ Provide actionable error messages

Monitoring

- ✔ Log all tool invocations
 - ✔ Track response times (p50, p95, p99)
 - ✔ Monitor error rates
 - ✔ Set up alerts for anomalies
-

Summary

As a solution architect, understanding these patterns allows you to:

1. **Design scalable systems** that grow with demand
2. **Ensure security** at every layer
3. **Optimize performance** through caching and pooling
4. **Handle failures gracefully** with circuit breakers and retries
5. **Integrate multiple systems** seamlessly

Start with the simplest pattern that meets your needs, then evolve as requirements grow.

Remember: The best architecture is one that solves your specific problem while remaining maintainable and extensible.