Research Documentation: MCP Discovery & Kagent Integration

Research Documentation: MCP Discovery & Kagent Integration

Executive Summary

This research document provides comprehensive analysis of building a Google Python ADK agent capable of autodiscovering MCP servers and converting configurations to Kagent format. The research covers protocol specifications, authentication frameworks, network discovery patterns, and production deployment architectures.

🔬 Research Methodology

Scope of Investigation

- Google Python Agent Development Kit (ADK) capabilities and architecture patterns
- MCP (Model Context Protocol) v1.0+ specification and discovery mechanisms
- Network service discovery protocols (mDNS, SSDP, ARP scanning)
- OAuth 2.1 + PKCE authentication requirements for MCP
- Kagent schema requirements and CRD structures
- Python 3.12+ implementation patterns for production deployment

Key Research Questions

1. How can Google ADK v1.0+ be leveraged for hierarchical agent architectures?
2. What are the technical requirements for MCP server autodiscovery?
3. How should OAuth 2.1 + PKCE authentication be implemented for automated systems?
4. What schema mapping strategies preserve semantic integrity between MCP and Kagent?
5. Which Python patterns optimize network discovery performance and reliability?

🏗️ Google Python ADK Architecture Analysis

Framework Capabilities

Core Agent Types

- **LlmAgent**: LLM-powered reasoning with model integration
- **BaseAgent**: Custom implementations with full control
- **Workflow Agents**: Sequential, Parallel, and Loop orchestration patterns

The ADK follows an event-driven, modular architecture supporting hierarchical multi-agent systems where parent agents coordinate sub-agents through structured event communication.

Production Integration Patterns

- **Vertex AI Agent Engine**: Fully managed runtime with auto-scaling capabilities
- **Vertex AI Model Garden**: Access to 200+ models including Gemini 2.0 Flash
- **Cloud Service Integration**: Direct BigQuery, AlloyDB, and Cloud Storage connectors
- **Authentication**: Application Default Credentials for development, Service Account integration for production

Security Framework

- OAuth 2.0, HTTPBearer, APIKey, and OpenID Connect schemes
- OpenAPI 3.0 compatibility with comprehensive validation

- Vertex Gemini Enterprise API for sandboxed code execution
- Callback systems (before_model_callback, before_tool_callback) for security controls

Implementation Recommendations

Agent Hierarchy Design

```
OrchestrationAgent (Parent)
├── DiscoveryAgent (Child) - Network scanning and classification
├── AuthenticationAgent (Child) - OAuth 2.1 credential management
├── ConversionAgent (Child) - Schema transformation pipeline
└── MCPClientAgent (Child) - Protocol communication
```

Event Communication Pattern

- Type-safe inter-agent messaging using Pydantic models
- Circuit breaker patterns for external service calls
- Structured logging with correlation IDs for distributed tracing

🌐 MCP Protocol Specification Analysis

Protocol Foundation

Transport Layer

- **JSON-RPC 2.0**: Core messaging protocol for all MCP communication
- **STDIO**: Local process execution for development and testing
- **Server-Sent Events (SSE)**: Remote connections with real-time updates
- **HTTP/WebSocket**: API integration for production deployments

Capability Types

- **Tools**: Executable functions with JSON Schema parameter definitions
- **Resources**: Data sources with URI-based access patterns
- **Prompts**: Template systems for dynamic content generation
- **Sampling**: Advanced model interaction capabilities

Discovery Architecture

Current Limitations

MCP v1.0+ relies primarily on manual configuration through JSON files. Network-based discovery requires custom implementation using standard protocols.

Proposed Discovery Methods

- **mDNS/DNS-SD**: Service announcement with `_mcp._tcp.local` service type
- **SSDP/UPnP**: Windows-compatible discovery for enterprise environments
- **Custom Registry**: Centralized service registration with health checking

Discovery Metadata Structure

```
MCPServiceRecord:
    service_name: str
    transport_type: Literal["stdio", "sse", "http", "websocket"]
    endpoint_url: Optional[HttpUrl]
    capabilities: List[MCPCapability]
    auth_requirements: AuthenticationMetadata
    version: str
```

```
    health_check_url: Optional[HttpUrl]
```

🔐 OAuth 2.1 + PKCE Authentication Analysis

Protocol Requirements

OAuth 2.1 Compliance

- **PKCE (Proof Key for Code Exchange)**: Mandatory for all client types per RFC 7636
- **Dynamic Client Registration**: RFC 7591 for automated client setup
- **Resource Indicators**: RFC 8707 for proper token scoping
- **Short-lived Tokens**: 15-60 minute access token lifetime

Security Controls

- Exact redirect URI validation (no pattern matching)
- Audience validation with no token passthrough
- State parameter for CSRF protection
- Secure random generation for PKCE challenge

Implementation Architecture

Authentication Flow

1. **Discovery Phase**: `/.well-known/oauth-protected-resource` endpoint discovery
2. **Metadata Retrieval**: Authorization server metadata via RFC 8414
3. **Client Registration**: Dynamic registration with capability declaration
4. **Token Exchange**: PKCE-enabled authorization code flow
5. **Token Management**: Automatic refresh with rotation

Credential Storage Strategy

- Encrypted at-rest storage for refresh tokens
- Memory-only storage for access tokens
- Automatic credential rotation on expiration
- Fallback authentication strategies for offline scenarios

🕸️ Network Discovery Implementation Analysis

Primary Technology: python-zeroconf

Technical Specifications

- Pure Python implementation with optional Cython extensions
- IPv4 and IPv6 support with no external dependencies
- Async variants available through aiozeroconf wrapper
- Performance: 100+ service discovery within 30 seconds on typical networks

Implementation Pattern

```python
class MCPDiscoveryService:
    def __init__(self):
        self.zeroconf = AsyncZeroconf()
        self.browser = AsyncServiceBrowser(
            zeroconf=self.zeroconf,
            type_="_mcp._tcp.local.",
```

```
            handlers=[self._on_service_state_change]
        )

    async def _on_service_state_change(
        self,
        zeroconf: AsyncZeroconf,
        service_type: str,
        name: str,
        state_change: ServiceStateChange
    ) -> None:
        # Service discovery event handling
        pass
```

Complementary Discovery Protocols

SSDP (Simple Service Discovery Protocol)

- UPnP compatibility for Windows environments
- UDP multicast discovery with XML device descriptions
- Essential for enterprise networks with mixed operating systems

ARP-based Network Scanning

- Direct Layer 2 enumeration for completeness
- Identifies potential MCP servers without service announcement
- Requires privilege escalation for raw socket access

Performance Optimization Strategies

Concurrent Discovery

- Asyncio semaphore-based concurrency control (default: 50 concurrent operations)
- Connection pooling with aiohttp.TCPConnector (100 total, 20 per host)
- Result caching with 30-60 second TTL for discovered services

Resource Management

- Memory usage under 50MB for typical home networks (100+ devices)
- CPU usage under 20% during active discovery phases
- Proper cleanup with asyncio context managers

🔄 Schema Mapping Architecture

Kagent CRD Structure Analysis

Kubernetes Integration

- Custom Resource Definitions with `apiVersion: kagent.dev/v1alpha1`
- Agent and Tool resources with structured metadata
- Built-in support for tool categorization and risk assessment

Required Fields

```
apiVersion: kagent.dev/v1alpha1
kind: Tool
metadata:
  name: mcp-tool-identifier
  labels:
    kagent.dev/category: "api|file_system|database|external_service|compute|utility"
```

```
    kagent.dev/risk-level: "low|medium|high"
spec:
  type: mcp  # or builtin, http
  description: "Tool description"
  schema:
    # JSON Schema for parameters
  mcpConfig:
    serverEndpoint: "https://mcp-server.example.com"
    toolName: "original-mcp-tool-name"
```

MCP Tool Schema Structure

JSON Schema Format

```
MCPTool:
    name: str  # Required
    description: str  # Required
    inputSchema: Dict[str, Any]  # JSON Schema Draft 7
    outputSchema: Optional[Dict[str, Any]]
    annotations: Optional[MCPAnnotations]

MCPAnnotations:
    readOnlyHint: Optional[bool]
    destructiveHint: Optional[bool]
    idempotentHint: Optional[bool]
```

Bidirectional Conversion Strategy

MCP → Kagent Conversion

1. **Name Sanitization**: Convert MCP names to Kubernetes-compliant identifiers
2. **Schema Transformation**: Wrap single-property schemas in object containers
3. **Metadata Mapping**: Transform MCP annotations to Kagent labels and categories
4. **CRD Generation**: Create properly formatted Kubernetes resources

Kagent → MCP Conversion

1. **Schema Extraction**: Extract JSON Schema from Kagent CRD spec
2. **Metadata Preservation**: Map Kagent categories to MCP annotations
3. **Validation**: Ensure MCP compliance with protocol specifications
4. **Tool Registration**: Format for MCP server tool registration

Quality Assurance

- 100% semantic preservation for core capabilities
- Zero data loss during bidirectional conversion
- JSON Schema Draft 7 validation for all transformations
- Property-based testing with Hypothesis for edge case discovery

🚀 Python 3.12+ Implementation Patterns

Performance Optimizations

AsyncIO Enhancements

- 75% performance improvement in asyncio socket operations
- Eager task execution reducing scheduling overhead

- Optimized connection pooling with persistent connections

Modern Patterns

```python
# Structured concurrency with TaskGroup
async def discover_services(self) -> List[MCPService]:
    async with asyncio.TaskGroup() as tg:
        mdns_task = tg.create_task(self._discover_mdns())
        ssdp_task = tg.create_task(self._discover_ssdp())
        arp_task = tg.create_task(self._discover_arp())

    return self._merge_results(mdns_task.result(), ssdp_task.result(), arp_task.result())

# Robust timeout handling
async def authenticate_with_server(self, server: MCPServer) -> AccessToken:
    async with asyncio.timeout(30.0):
        return await self._oauth_flow(server)
```

Type Safety with Pydantic V2

Configuration Management

```python
class MCPDiscoveryConfig(BaseModel):
    model_config = ConfigDict(
        str_strip_whitespace=True,
        validate_assignment=True,
        use_enum_values=True
    )

    discovery_timeout: int = Field(default=30, ge=5, le=300)
    max_concurrent_discoveries: int = Field(default=50, ge=1, le=200)
    cache_ttl_seconds: int = Field(default=60, ge=10, le=3600)

    @field_validator('discovery_timeout')
    @classmethod
    def validate_timeout(cls, v: int) -> int:
        if v < 5:
            raise ValueError('Discovery timeout must be at least 5 seconds')
        return v
```

Package Management with UV

Performance Benefits

- 10-100x faster installation compared to pip
- Rust-powered dependency resolution with conflict detection
- Unified replacement for pip, pip-tools, and virtualenv

Project Structure

```toml
[project]
name = "mcp-vacuum"
version = "0.1.0"
requires-python = ">=3.12"
dependencies = [
    "google-adk>=1.0.0",
    "pydantic>=2.0.0",
    "asyncio-zeroconf>=0.3.0",
    "httpx>=0.25.0",
```

```
    "typer>=0.9.0"
]

[project.optional-dependencies]
dev = [
    "pytest>=7.4.0",
    "pytest-asyncio>=0.21.0",
    "pytest-cov>=4.1.0",
    "ruff>=0.1.0",
    "mypy>=1.5.0"
]
```

🏭 Production Deployment Architecture

Google Cloud Integration

Vertex AI Agent Engine

- Fully managed runtime with automatic scaling
- Built-in monitoring and observability
- Integrated security with IAM and VPC controls

Service Integration Strategy

- **Cloud Storage**: Configuration caching and service registry
- **BigQuery**: Discovery analytics and performance metrics
- **Cloud Monitoring**: Real-time alerting and dashboard visualization
- **Cloud Logging**: Structured log aggregation with correlation

Security Implementation

Network Security

- VPC-native networking with private service connect
- Firewall rules limiting MCP server access to specific ports
- Network policy enforcement for discovered services

Identity and Access Management

- Service Account with minimal required permissions
- Workload Identity for Kubernetes deployment
- Credential rotation automation with Cloud KMS integration

Audit and Compliance

- Comprehensive audit logging for all discovery and conversion operations
- Data residency controls for multi-region deployments
- SOC 2 Type II compliance for enterprise deployments

📊 Performance and Reliability Requirements

Service Level Objectives (SLOs)

Discovery Performance

- 95% of discovery operations complete within 30 seconds
- Support for 500+ concurrent service evaluations
- Memory usage remains below 200MB during peak operations

Authentication Reliability

- 99.9% OAuth flow success rate for valid configurations
- Token refresh operations complete within 5 seconds
- Credential rotation with zero downtime

Schema Conversion Accuracy

- 100% semantic preservation for MCP core capabilities
- Conversion operations complete within 100ms for typical configurations
- Zero data loss during bidirectional transformations

Error Handling and Recovery

Circuit Breaker Patterns

- Fail-fast for unreachable MCP servers
- Exponential backoff with jitter for retry operations
- Graceful degradation when discovery services are unavailable

Monitoring and Alerting

- Real-time metrics for discovery success rates
- Performance dashboards with latency percentiles
- Automated alerting for authentication failures

🎯 Implementation Recommendations

Development Priorities

Phase 1: Foundation (Weeks 1-2)

1. Project setup with UV and modern Python tooling
2. Core Pydantic models for MCP and Kagent schemas
3. Basic Google ADK agent framework implementation

Phase 2: Core Functionality (Weeks 3-5)

1. Multi-protocol network discovery implementation
2. OAuth 2.1 + PKCE authentication system
3. MCP JSON-RPC 2.0 client with full transport support

Phase 3: Integration (Weeks 6-8)

1. Bidirectional schema conversion engine
2. ADK agent orchestration and lifecycle management
3. Comprehensive testing suite with real-world scenarios

Phase 4: Production (Weeks 9-12)

1. Google Cloud deployment automation
2. Monitoring, alerting, and observability
3. Performance optimization and security hardening

Technical Debt Mitigation

Code Quality Standards

- 95%+ test coverage for core logic components

- Type safety with mypy strict mode
- Comprehensive documentation with architectural decision records

Operational Excellence

- Infrastructure as Code with Terraform
- CI/CD pipeline with automated security scanning
- Disaster recovery procedures with documented runbooks

📚 References and Further Reading

Primary Sources

- **Google ADK Documentation**: https://google.github.io/adk-docs/
- **MCP Specification**: https://modelcontextprotocol.io/specification/
- **OAuth 2.1 RFC**: https://datatracker.ietf.org/doc/draft-ietf-oauth-v2-1/
- **PKCE RFC 7636**: https://tools.ietf.org/html/rfc7636
- **Dynamic Client Registration RFC 7591**: https://tools.ietf.org/html/rfc7591

Technical Standards

- **JSON-RPC 2.0**: https://www.jsonrpc.org/specification
- **mDNS RFC 6763**: https://tools.ietf.org/html/rfc6763
- **DNS-SD RFC 6763**: https://tools.ietf.org/html/rfc6763
- **SSDP UPnP Specification**: http://upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.0.pdf

Implementation Guides

- **Python AsyncIO Best Practices**: https://docs.python.org/3/library/asyncio.html
- **Pydantic V2 Documentation**: https://docs.pydantic.dev/latest/
- **pytest-asyncio Patterns**: https://pytest-asyncio.readthedocs.io/

Research conducted: June 24, 2025
Document version: 1.0.0
Next review: July 24, 2025