Schema Conversion & Mapping Guide

Schema Conversion & Mapping Guide

Bidirectional MCP ↔ Kagent Transformation

Intelligent schema conversion between Model Context Protocol (MCP) and Kagent formats with semantic preservation and validation.

Core Conversion Engine

Schema Converter Implementation

```python
from typing import Any, Dict, List
from pydantic import BaseModel, Field
import re

class SchemaConverter:
    """Bidirectional schema conversion with semantic preservation."""

    def __init__(self) -> None:
        self.field_mappings = self._load_field_mappings()
        self.type_mappings = self._load_type_mappings()

    def mcp_to_kagent(self, mcp_tool: MCPTool) -> KagentTool:
        """Convert MCP tool to Kagent CRD format."""

        return KagentTool(
            apiVersion="tools.kagent.ai/v1",
            kind="Tool",
            metadata=self._convert_metadata(mcp_tool),
            spec=self._convert_tool_spec(mcp_tool)
        )

    def kagent_to_mcp(self, kagent_tool: KagentTool) -> MCPTool:
        """Convert Kagent tool back to MCP format."""

        return MCPTool(
            name=self._extract_original_name(kagent_tool.metadata),
            description=kagent_tool.spec.description,
            input_schema=self._convert_k8s_to_json_schema(
                kagent_tool.spec.parameters
            ),
            output_schema=self._convert_k8s_to_json_schema(
                kagent_tool.spec.output_schema
            ) if kagent_tool.spec.output_schema else None
        )
```

Metadata Conversion

```python
def _convert_metadata(self, mcp_tool: MCPTool) -> KagentMetadata:
    """Convert MCP tool metadata to Kubernetes format."""

    # Sanitize name for Kubernetes compliance
```

```
    k8s_name = self._sanitize_k8s_name(mcp_tool.name)

    return KagentMetadata(
        name=k8s_name,
        labels={
            "mcp.source": "mcp-vacuum",
            "mcp.version": "1.0",
            "mcp.original-name": mcp_tool.name,
            "category": self._categorize_tool(mcp_tool),
            "risk-level": self._assess_risk_level(mcp_tool)
        },
        annotations={
            "mcp.vacuum/original-name": mcp_tool.name,
            "mcp.vacuum/discovered-at": datetime.utcnow().isoformat(),
            "mcp.vacuum/server-endpoint": mcp_tool.server_endpoint,
            "description": mcp_tool.description[:250] + "..."
            if len(mcp_tool.description) > 250 else mcp_tool.description
        }
    )

def _sanitize_k8s_name(self, name: str) -> str:
    """Sanitize name for Kubernetes compliance.

    Kubernetes names must:
    - Start and end with alphanumeric characters
    - Contain only lowercase letters, numbers, and hyphens
    - Be at most 63 characters long
    """
    # Convert to lowercase and replace invalid chars with hyphens
    sanitized = re.sub(r'[^a-z0-9-]', '-', name.lower())

    # Ensure starts and ends with alphanumeric
    sanitized = re.sub(r'^[^a-z0-9]+|[^a-z0-9]+$', '', sanitized)

    # Remove consecutive hyphens
    sanitized = re.sub(r'-+', '-', sanitized)

    # Truncate to 63 characters
    return sanitized[:63]
```

JSON Schema to Kubernetes CRD Mapping

Schema Structure Conversion

```
def _convert_json_schema_to_k8s(
    self,
    json_schema: Dict[str, Any]
) -> Dict[str, Any]:
    """Convert JSON Schema Draft 7 to Kubernetes CRD OpenAPI v3 schema."""

    if not json_schema:
        return {}

    # Deep copy to avoid modifying original
    k8s_schema = json.loads(json.dumps(json_schema))

    # Apply Kubernetes-specific transformations
    k8s_schema = self._transform_schema_structure(k8s_schema)
```

```
    k8s_schema = self._apply_k8s_naming_conventions(k8s_schema)
    k8s_schema = self._add_k8s_validation_rules(k8s_schema)

    return k8s_schema

def _transform_schema_structure(self, schema: Dict[str, Any]) -> Dict[str, Any]:
    """Transform JSON Schema structure for Kubernetes compatibility."""

    # Remove JSON Schema specific keywords not supported in OpenAPI v3
    unsupported_keywords = [
        "$schema", "$id", "definitions", "$ref", "const"
    ]

    for keyword in unsupported_keywords:
        schema.pop(keyword, None)

    # Transform definitions to components/schemas
    if "definitions" in schema:
        # Move definitions to separate section (handled at CRD level)
        definitions = schema.pop("definitions")
        schema["x-kubernetes-definitions"] = definitions

    # Transform recursive schemas
    if "properties" in schema:
        schema["properties"] = {
            key: self._transform_schema_structure(value)
            for key, value in schema["properties"].items()
        }

    # Transform array items
    if "items" in schema:
        schema["items"] = self._transform_schema_structure(schema["items"])

    return schema
```

Field Name Transformation

```
def _apply_k8s_naming_conventions(
    self,
    schema: Dict[str, Any]
) -> Dict[str, Any]:
    """Apply Kubernetes field naming conventions."""

    if "properties" in schema:
        new_properties = {}

        for field_name, field_schema in schema["properties"].items():
            # Convert field names to camelCase (Kubernetes standard)
            k8s_field_name = self._to_camel_case(field_name)

            # Recursively apply to nested objects
            new_properties[k8s_field_name] = self._apply_k8s_naming_conventions(
                field_schema
            )

        schema["properties"] = new_properties

    return schema
```

```
22
23  def _to_camel_case(self, snake_str: str) -> str:
24      """Convert snake_case to camelCase."""
25      components = snake_str.split('_')
26      return components[0] + ''.join(word.capitalize() for word in components[1:])
```

Validation Pipeline

Multi-Stage Validation

```python
from enum import Enum

class ValidationSeverity(str, Enum):
    ERROR = "error"
    WARNING = "warning"
    INFO = "info"

class ValidationIssue(BaseModel):
    severity: ValidationSeverity
    message: str
    field_path: str
    suggested_fix: str | None = None

class ValidationResult(BaseModel):
    is_valid: bool
    issues: List[ValidationIssue]
    schema_hash: str

    @property
    def has_errors(self) -> bool:
        return any(issue.severity == ValidationSeverity.ERROR for issue in self.issues)

class ValidationPipeline:
    """Multi-stage schema validation with detailed reporting."""

    async def validate_conversion(
        self,
        original: MCPTool,
        converted: KagentTool
    ) -> ValidationResult:
        """Validate schema conversion preserves semantics."""

        issues: List[ValidationIssue] = []

        # Stage 1: Structural validation
        issues.extend(await self._validate_structure(converted))

        # Stage 2: Semantic preservation
        issues.extend(await self._validate_semantics(original, converted))

        # Stage 3: Kubernetes compatibility
        issues.extend(await self._validate_k8s_compatibility(converted))

        # Stage 4: Field mapping validation
        issues.extend(await self._validate_field_mappings(original, converted))

        schema_hash = self._compute_schema_hash(converted)

```

```
        return ValidationResult(
            is_valid=not any(issue.severity == ValidationSeverity.ERROR for issue in issues),
            issues=issues,
            schema_hash=schema_hash
        )
```

Semantic Preservation Validation

```python
async def _validate_semantics(
    self,
    original: MCPTool,
    converted: KagentTool
) -> List[ValidationIssue]:
    """Validate semantic preservation during conversion."""

    issues = []

    # Check if essential fields are preserved
    if not self._is_semantically_equivalent(
        original.input_schema,
        converted.spec.parameters
    ):
        issues.append(ValidationIssue(
            severity=ValidationSeverity.ERROR,
            message="Input schema semantics not preserved",
            field_path="spec.parameters",
            suggested_fix="Review field mappings and type conversions"
        ))

    # Check required fields preservation
    original_required = set(original.input_schema.get("required", []))
    converted_required = set(self._extract_required_fields(converted.spec.parameters))

    missing_required = original_required - converted_required
    if missing_required:
        issues.append(ValidationIssue(
            severity=ValidationSeverity.WARNING,
            message=f"Required fields not preserved: {missing_required}",
            field_path="spec.parameters.required",
            suggested_fix="Add missing required field validations"
        ))

    # Check field types preservation
    type_issues = self._validate_type_preservation(original, converted)
    issues.extend(type_issues)

    return issues

def _is_semantically_equivalent(
    self,
    original_schema: Dict[str, Any],
    converted_schema: Dict[str, Any]
) -> bool:
    """Check if schemas are semantically equivalent."""

    # Compare essential structure
    original_props = set(original_schema.get("properties", {}).keys())
    converted_props = set(self._extract_property_names(converted_schema))
```

```
    # Allow for reasonable field name transformations
    normalized_original = {self._normalize_field_name(p) for p in original_props}
    normalized_converted = {self._normalize_field_name(p) for p in converted_props}

    # Semantic equivalence if > 80% field overlap
    overlap = len(normalized_original & normalized_converted)
    total_fields = len(normalized_original | normalized_converted)

    return total_fields == 0 or (overlap / total_fields) >= 0.8
```

Tool Categorization & Risk Assessment

Intelligent Categorization

```python
from typing import Set

class ToolCategory(str, Enum):
    FILE_OPERATIONS = "file-operations"
    NETWORK_ACCESS = "network-access"
    SYSTEM_COMMANDS = "system-commands"
    DATA_PROCESSING = "data-processing"
    API_INTEGRATION = "api-integration"
    COMPUTATION = "computation"
    UNKNOWN = "unknown"

class RiskLevel(str, Enum):
    LOW = "low"
    MEDIUM = "medium"
    HIGH = "high"
    CRITICAL = "critical"

class ToolAnalyzer:
    """Intelligent tool categorization and risk assessment."""

    def __init__(self) -> None:
        self.category_keywords = self._load_category_keywords()
        self.risk_indicators = self._load_risk_indicators()

    def categorize_tool(self, tool: MCPTool) -> ToolCategory:
        """Categorize tool based on name, description, and schema."""

        text_content = f"{tool.name} {tool.description}".lower()

        # Check schema for operation types
        schema_indicators = self._extract_schema_indicators(tool.input_schema)
        text_content += " " + " ".join(schema_indicators)

        # Score each category
        category_scores = {}
        for category, keywords in self.category_keywords.items():
            score = sum(1 for keyword in keywords if keyword in text_content)
            category_scores[category] = score

        # Return highest scoring category or UNKNOWN
        if category_scores:
            best_category = max(category_scores.items(), key=lambda x: x[1])
            return ToolCategory(best_category[0]) if best_category[1] > 0 else ToolCategory.UNKNOWN
```

```
        return ToolCategory.UNKNOWN

    def assess_risk_level(self, tool: MCPTool) -> RiskLevel:
        """Assess security risk level of tool."""

        risk_score = 0
        text_content = f"{tool.name} {tool.description}".lower()

        # Check for high-risk keywords
        for risk_indicator, score in self.risk_indicators.items():
            if risk_indicator in text_content:
                risk_score += score

        # Check schema for risky parameters
        schema_risk = self._assess_schema_risk(tool.input_schema)
        risk_score += schema_risk

        # Convert score to risk level
        if risk_score >= 10:
            return RiskLevel.CRITICAL
        elif risk_score >= 7:
            return RiskLevel.HIGH
        elif risk_score >= 4:
            return RiskLevel.MEDIUM
        else:
            return RiskLevel.LOW
```

Conversion Metadata Preservation

Metadata Tracking

```
class ConversionMetadata(BaseModel):
    """Metadata about the conversion process."""

    original_tool_name: str
    conversion_timestamp: datetime
    conversion_version: str
    semantic_score: float = Field(ge=0.0, le=1.0)
    validation_results: ValidationResult
    field_mappings: Dict[str, str]

    class Config:
        json_encoders = {
            datetime: lambda v: v.isoformat()
        }

class MetadataPreserver:
    """Preserve conversion metadata for traceability."""

    def create_conversion_metadata(
        self,
        original: MCPTool,
        converted: KagentTool,
        validation: ValidationResult
    ) -> ConversionMetadata:
        """Create comprehensive conversion metadata."""

```

```
        field_mappings = self._extract_field_mappings(original, converted)
        semantic_score = self._calculate_semantic_score(original, converted)

        return ConversionMetadata(
            original_tool_name=original.name,
            conversion_timestamp=datetime.utcnow(),
            conversion_version="1.0.0",
            semantic_score=semantic_score,
            validation_results=validation,
            field_mappings=field_mappings
        )

    def embed_metadata_in_kagent(
        self,
        kagent_tool: KagentTool,
        metadata: ConversionMetadata
    ) -> KagentTool:
        """Embed conversion metadata in Kagent tool annotations."""

        # Add conversion metadata to annotations
        kagent_tool.metadata.annotations.update({
            "mcp.vacuum/conversion-metadata": metadata.json(),
            "mcp.vacuum/semantic-score": str(metadata.semantic_score),
            "mcp.vacuum/conversion-version": metadata.conversion_version
        })

        return kagent_tool
```

Performance Optimization

Batch Conversion

```
class BatchConverter:
    """Optimized batch schema conversion."""

    def __init__(self, max_concurrent: int = 10) -> None:
        self.semaphore = asyncio.Semaphore(max_concurrent)
        self.converter = SchemaConverter()

    async def convert_batch(
        self,
        tools: List[MCPTool]
    ) -> List[ConversionResult]:
        """Convert multiple tools concurrently."""

        tasks = [
            asyncio.create_task(self._convert_with_semaphore(tool))
            for tool in tools
        ]

        results = await asyncio.gather(*tasks, return_exceptions=True)

        return [
            result for result in results
            if isinstance(result, ConversionResult)
        ]

    async def _convert_with_semaphore(self, tool: MCPTool) -> ConversionResult:
```

```
        """Convert single tool with concurrency control."""
        async with self.semaphore:
            try:
                converted = self.converter.mcp_to_kagent(tool)
                validation = await self.converter.validate_conversion(tool, converted)

                return ConversionResult(
                    original=tool,
                    converted=converted,
                    validation=validation,
                    success=True
                )
            except Exception as e:
                return ConversionResult(
                    original=tool,
                    converted=None,
                    validation=None,
                    success=False,
                    error=str(e)
                )
```

Next Steps

1. **Implement Core Converter**: Start with basic MCP to Kagent conversion
2. **Add Validation Pipeline**: Implement multi-stage validation
3. **Tool Categorization**: Add intelligent categorization and risk assessment
4. **Metadata Preservation**: Implement conversion traceability
5. **Performance Optimization**: Add batch processing capabilities