Schema and API Specification

OpenRPC and JSON Schema Integration for Service Function Architecture

---

Executive Summary

This document defines the schema and API specification standards for the Service Function Architecture platform. The specification uses OpenRPC for service function definitions and JSON Schema for data type specifications, creating a unified type system that spans service functions, database schemas, UI models, and cross-language code generation.

Key Components:

- **OpenRPC Specifications**: Define service function interfaces and method signatures
- **JSON Schema Integration**: Provide rich type definitions for all data structures
- **Signature Compatibility System**: Enable safe schema evolution and version management
- **Cross-Language Code Generation**: Generate native function stubs and type definitions
- **Unified Type System**: Single source of truth for all platform data types

Benefits:

- Complete type safety across all platform components
- Automated compatibility checking for schema evolution
- Native code generation for multiple programming languages
- Single specification format for functions, data, and interfaces
- Coordinated updates across service functions, database schemas, and UI models

---

OpenRPC Foundation

Core Specification Structure

The platform uses OpenRPC 1.2.6 as the foundation for all service function definitions. Each service library exports a complete OpenRPC specification that describes its available functions, parameter types, and return values.

Standard Service Specification Format:

json


```
{
 "openrpc": "1.2.6",
 "info": {
  "title": "UserManagement",
  "version": "1.2.4",
  "description": "User management service functions",
  "contact": {
   "name": "Development Team",
   "email": "dev@company.com"
  }
 },
 "servers": [
  {
   "name": "production",
   "url": "mqtt://production-cluster/service/UserManagement"
  },
  {
   "name": "development",
   "url": "mqtt://localhost/service/UserManagement"
  }
 ],
 "methods": [
  {
   "name": "createUser",
   "summary": "Create a new user account",
   "description": "Creates a new user with the provided information and returns the user ID",
   "params": [
    {
     "name": "request",
     "description": "User creation request containing user details",
     "required": true,
     "schema": {
      "$ref": "#/components/schemas/CreateUserRequest"
     }
    }
   ],
   "result": {
    "name": "CreateUserResponse",
    "description": "Response containing the created user ID and success status",
    "schema": {
     "$ref": "#/components/schemas/CreateUserResponse"
    }
   },
```

```
    "errors": [
     {
      "code": -32001,
      "message": "Validation Error",
      "data": {
       "$ref": "#/components/schemas/ValidationError"
      }
     },
     {
      "code": -32002,
      "message": "Business Logic Error",
      "data": {
       "$ref": "#/components/schemas/BusinessLogicError"
      }
     }
    ]
   }
  ],
  "components": {
   "schemas": {
    "CreateUserRequest": {
     "type": "object",
     "properties": {
      "email": {
       "type": "string",
       "format": "email",
       "description": "User's email address - must be unique"
      },
      "name": {
       "type": "string",
       "minLength": 1,
       "maxLength": 100,
       "description": "User's full name"
      },
      "department": {
       "type": "string",
       "description": "Optional department assignment"
      },
      "profile": {
       "$ref": "#/components/schemas/UserProfile"
      }
     },
     "required": ["email", "name"],
     "additionalProperties": false
    },
    "CreateUserResponse" {
```

```
    "CreateUserResponse": {
     "type": "object",
     "properties": {
      "userId": {
       "type": "string",
       "format": "uuid",
       "description": "Unique identifier for the created user"
      },
      "success": {
       "type": "boolean",
       "description": "Indicates whether the operation succeeded"
      },
      "user": {
       "$ref": "#/components/schemas/User",
       "description": "Complete user object with all populated fields"
      }
     },
     "required": ["userId", "success"],
     "additionalProperties": false
    }
   }
  }
}
```

Method Definition Standards

Parameter Conventions:

- Service functions should accept a single `request` parameter containing all input data
- Use JSON Schema `$ref` to reference reusable type definitions
- Mark parameters as `required: true` when mandatory
- Provide clear descriptions for all parameters and their validation rules

Response Conventions:

- All responses should include a `success` boolean field
- Use consistent error codes for different error types (-32001 for validation, -32002 for business logic)
- Include complete object references when returning entity data
- Provide meaningful descriptions for all response fields

Error Handling:

- Define standard error codes and formats across all services
- Use JSON Schema for error data structures
- Include validation context in error responses
- Maintain consistency in error message formats

JSON Schema Integration

Rich Type Definitions

JSON Schema provides the foundation for all data type definitions within the platform. These schemas are used across service functions, database collections, UI models, and generated code.

Complex Type Example:

json

```
{
 "User": {
  "type": "object",
  "title": "User Entity",
  "description": "Complete user profile with all associated data",
  "properties": {
   "id": {
    "type": "string",
    "format": "uuid",
    "description": "Unique user identifier"
   },
   "email": {
    "type": "string",
    "format": "email",
    "description": "User's email address"
   },
   "name": {
    "type": "string",
    "minLength": 1,
    "maxLength": 100,
    "description": "User's full name"
   },
   "department": {
    "type": "string",
    "description": "User's department assignment"
   },
   "profile": {
    "$ref": "#/components/schemas/UserProfile"
   },
   "preferences": {
    "$ref": "#/components/schemas/UserPreferences"
   },
   "roles": {
    "type": "array",
    "items": {
     "$ref": "#/components/schemas/Role"
    },
    "description": "User's assigned roles"
   },
   "metadata": {
    "type": "object",
    "properties": {
     "createdAt": {
```

```
        "type": "string",
        "format": "date-time",
        "description": "Account creation timestamp"
       },
       "updatedAt": {
        "type": "string",
        "format": "date-time",
        "description": "Last profile update timestamp"
       },
       "lastLogin": {
        "type": "string",
        "format": "date-time",
        "description": "Most recent login timestamp"
       }
      },
      "required": ["createdAt"]
     }
    },
    "required": ["id", "email", "name", "metadata"],
    "additionalProperties": false
   },
   "UserProfile": {
    "type": "object",
    "description": "Extended user profile information",
    "properties": {
     "avatar": {
      "type": "string",
      "format": "uri",
      "description": "URL to user's avatar image"
     },
     "bio": {
      "type": "string",
      "maxLength": 500,
      "description": "User's biographical information"
     },
     "location": {
      "type": "string",
      "description": "User's geographic location"
     },
     "timezone": {
      "type": "string",
      "pattern": "^[A-Za-z]+/[A-Za-z_]+$",
      "description": "User's timezone (e.g., America/New_York)"
     }
    },
    "additionalProperties" false
```

```
    "additionalProperties": false
  },
  "UserPreferences": {
    "type": "object",
    "description": "User's application preferences and settings",
    "properties": {
      "theme": {
        "type": "string",
        "enum": ["light", "dark", "auto"],
        "default": "auto",
        "description": "UI theme preference"
      },
      "language": {
        "type": "string",
        "pattern": "^[a-z]{2}(-[A-Z]{2})?$",
        "default": "en",
        "description": "Preferred language (ISO 639-1 format)"
      },
      "notifications": {
        "type": "object",
        "properties": {
          "email": {
            "type": "boolean",
            "default": true,
            "description": "Enable email notifications"
          },
          "push": {
            "type": "boolean",
            "default": true,
            "description": "Enable push notifications"
          },
          "sms": {
            "type": "boolean",
            "default": false,
            "description": "Enable SMS notifications"
          }
        }
      }
    },
    "additionalProperties": false
  }
}
```

Schema Composition Patterns

Inheritance and Composition:

```json
{
 "BaseEntity": {
  "type": "object",
  "properties": {
   "id": {"type": "string", "format": "uuid"},
   "createdAt": {"type": "string", "format": "date-time"},
   "updatedAt": {"type": "string", "format": "date-time"}
  },
  "required": ["id", "createdAt"]
 },
 "User": {
  "allOf": [
   {"$ref": "#/components/schemas/BaseEntity"},
   {
    "type": "object",
    "properties": {
     "email": {"type": "string", "format": "email"},
     "name": {"type": "string"}
    },
    "required": ["email", "name"]
   }
  ]
 }
}
```

Polymorphic Types:

json

```
{
 "PaymentMethod": {
  "oneOf": [
   {"$ref": "#/components/schemas/CreditCard"},
   {"$ref": "#/components/schemas/BankTransfer"},
   {"$ref": "#/components/schemas/DigitalWallet"}
  ],
  "discriminator": {
   "propertyName": "type",
   "mapping": {
    "credit_card": "#/components/schemas/CreditCard",
    "bank_transfer": "#/components/schemas/BankTransfer",
    "digital_wallet": "#/components/schemas/DigitalWallet"
   }
  }
 }
}
```

Signature Compatibility System

Caller and Callee Specifications

The platform maintains compatibility through explicit signature tracking for both callers (clients, UIs) and callees (service functions).

Caller Signature File (Client Requirements):

json

```
{
 "caller": {
  "name": "WebApplication",
  "version": "2.1.0",
  "description": "Main web application client"
 },
 "requiredServices": [
  {
   "service": "UserManagement",
   "methods": [
    {
     "name": "createUser",
     "params": [
      {
       "name": "request",
       "schema": {
        "type": "object",
        "properties": {
         "email": {"type": "string", "format": "email"},
         "name": {"type": "string"}
        },
        "required": ["email", "name"]
       }
      }
     ],
     "result": {
      "schema": {
       "type": "object",
       "properties": {
        "userId": {"type": "string"},
        "success": {"type": "boolean"}
       },
       "required": ["userId", "success"]
      }
     }
    }
   ]
  },
  {
   "service": "OrderManagement",
   "methods": [
    {
     "name": "createOrder",
```

```
      "params": [
       {
        "name": "request",
        "schema": {"$ref": "order-schemas.json#/CreateOrderRequest"}
       }
      ],
      "result": {
       "schema": {"$ref": "order-schemas.json#/CreateOrderResponse"}
      }
     }
    ]
   }
  ]
}
```

Callee Signature File (Service Capabilities):

json


```
{
 "callee": {
  "service": "UserManagement",
  "version": "1.2.4",
  "description": "User management service implementation"
 },
 "supportedMethods": [
  {
   "name": "createUser",
   "signatures": [
    {
     "version": "1.2.4",
     "params": [
      {
       "name": "request",
       "schema": {
        "type": "object",
        "properties": {
         "email": {"type": "string", "format": "email"},
         "name": {"type": "string"},
         "department": {"type": "string"}
        },
        "required": ["email", "name"]
       }
      }
     ],
     "result": {
      "schema": {
       "type": "object",
       "properties": {
        "userId": {"type": "string"},
        "success": {"type": "boolean"},
        "user": {"$ref": "#/components/schemas/User"}
       },
       "required": ["userId", "success"]
      }
     }
    },
    {
     "version": "1.2.0",
     "deprecated": true,
     "params": [
      {
```

```
          "name": "request",
          "schema": {
           "type": "object",
           "properties": {
            "email": {"type": "string", "format": "email"},
            "name": {"type": "string"}
           },
           "required": ["email", "name"]
          }
         }
        ],
        "result": {
         "schema": {
          "type": "object",
          "properties": {
           "userId": {"type": "string"},
           "success": {"type": "boolean"}
          },
          "required": ["userId", "success"]
         }
        }
       }
      ]
     }
    ]
}
```

Compatibility Checking Algorithm

Schema Compatibility Rules:

1. **Parameter Compatibility**: Callee must accept all required parameters from caller
2. **Response Compatibility**: Callee must provide all fields expected by caller
3. **Type Compatibility**: Field types must be compatible (covariant for responses, contravariant for parameters)
4. **Version Compatibility**: Semantic versioning rules for breaking vs non-breaking changes

Implementation Example:

csharp

```csharp
public class CompatibilityChecker
{
    public CompatibilityResult CheckCompatibility(CallerSignature caller, CalleeSignature callee)
    {
        var result = new CompatibilityResult();

        // Check method existence
        var calleeMethod = callee.SupportedMethods.FirstOrDefault(m => m.Name == caller.MethodName);
        if (calleeMethod == null)
        {
            result.AddError($"Method {caller.MethodName} not supported by callee");
            return result;
        }

        // Check parameter compatibility (contravariant)
        if (!AreParametersCompatible(caller.Params, callee.Params))
            return false;

        // Check result compatibility (covariant)
        if (!AreResultsCompatible(caller.Result, callee.Result))
            return false;

        return true;
    }

    private bool AreParametersCompatible(List<Parameter> callerParams, List<Parameter> calleeParams)
    {
        foreach (var callerParam in callerParams)
        {
            var calleeParam = calleeParams.FirstOrDefault(p => p.Name == callerParam.Name);

            if (calleeParam == null && callerParam.Required)
                return false; // Callee doesn't accept required parameter

            if (calleeParam != null && !IsSchemaCompatible(callerParam.Schema, calleeParam.Schema, contravariant
                return false; // Parameter type incompatible
        }

        // Check that callee doesn't require parameters caller doesn't provide
        foreach (var calleeParam in calleeParams.Where(p => p.Required))
        {
            if (!callerParams.Any(p => p.Name == calleeParam.Name))
                return false; // Callee requires parameter caller doesn't provide
```

```
        }

        return true;
    }
}
```

Migration Planning

Breaking Change Detection:

csharp

```
public class MigrationPlanner
{
    public MigrationPlan PlanServiceUpgrade(
        string serviceName,
        string currentVersion,
        string targetVersion,
        List<CallerSignature> allCallers)
    {
        var currentSpec = LoadServiceSpec(serviceName, currentVersion);
        var targetSpec = LoadServiceSpec(serviceName, targetVersion);

        var plan = new MigrationPlan();

        foreach (var caller in allCallers)
        {
            var currentCompatibility = CheckCompatibility(caller, currentSpec);
            var targetCompatibility = CheckCompatibility(caller, targetSpec);

            if (currentCompatibility.IsCompatible && !targetCompatibility.IsCompatible)
            {
                plan.AddBreakingChange(new BreakingChange
                {
                    AffectedCaller = caller.Name,
                    Service = serviceName,
                    IncompatibleMethods = targetCompatibility.IncompatibleMethods,
                    RequiredActions = GenerateRequiredActions(currentCompatibility, targetCompatibility)
                });
            }
        }

        return plan;
    }
}
```

Cross-Language Code Generation

Generated Function Stubs

The platform generates native function stubs in multiple programming languages from OpenRPC specifications, providing type-safe interfaces that hide the underlying messaging complexity.

C# Code Generation:

csharp

```csharp
// Generated from UserManagement.openrpc.json
using System;
using System.Threading.Tasks;
using Platform.Messaging;

namespace ServiceFunctions
{
  /// <summary>
  /// User management service functions
  /// Generated from UserManagement v1.2.4
  /// </summary>
  public static class UserManagement
  {
    private static readonly IMessageRouter _messageRouter = MessageRouter.Instance;

    /// <summary>
    /// Create a new user account
    /// </summary>
    /// <param name="request">User creation request containing user details</param>
    /// <returns>Response containing the created user ID and success status</returns>
    /// <exception cref="ValidationException">Thrown when input validation fails</exception>
    /// <exception cref="BusinessLogicException">Thrown when business rules are violated</exception>
    public static async Task<CreateUserResponse> CreateUser(CreateUserRequest request)
    {
      if (request == null)
        throw new ArgumentNullException(nameof(request));

      return await _messageRouter.CallAsync<CreateUserResponse>(
        serviceName: "UserManagement",
        methodName: "createUser",
        request: request
      );
    }

    /// <summary>
    /// Update an existing user's information
    /// </summary>
    public static async Task<UpdateUserResponse> UpdateUser(UpdateUserRequest request)
    {
      if (request == null)
        throw new ArgumentNullException(nameof(request));

      return await _messageRouter.CallAsync<UpdateUserResponse>(
```

```
            serviceName: "UserManagement",
            methodName: "updateUser",
            request: request
        );
    }
}

/// <summary>
/// User creation request containing user details
/// </summary>
public class CreateUserRequest
{
    /// <summary>
    /// User's email address - must be unique
    /// </summary>
    public string Email { get; set; }

    /// <summary>
    /// User's full name
    /// </summary>
    public string Name { get; set; }

    /// <summary>
    /// Optional department assignment
    /// </summary>
    public string Department { get; set; }

    /// <summary>
    /// Extended user profile information
    /// </summary>
    public UserProfile Profile { get; set; }
}

/// <summary>
/// Response containing the created user ID and success status
/// </summary>
public class CreateUserResponse
{
    /// <summary>
    /// Unique identifier for the created user
    /// </summary>
    public string UserId { get; set; }

    /// <summary>
    /// Indicates whether the operation succeeded
    /// </
```

```
        /// </summary>
        public bool Success { get; set; }

        /// <summary>
        /// Complete user object with all populated fields
        /// </summary>
        public User User { get; set; }
    }
}
```

TypeScript Code Generation:

typescript

```typescript
// Generated from UserManagement.openrpc.json
import { MessageRouter } from '@platform/messaging';

/**
 * User management service functions
 * Generated from UserManagement v1.2.4
 */
export class UserManagement {
 private static messageRouter = MessageRouter.getInstance();

 /**
  * Create a new user account
  * @param request User creation request containing user details
  * @returns Response containing the created user ID and success status
  * @throws ValidationError when input validation fails
  * @throws BusinessLogicError when business rules are violated
  */
 static async createUser(request: CreateUserRequest): Promise<CreateUserResponse> {
  if (!request) {
   throw new Error('Request parameter is required');
  }

  return await this.messageRouter.call<CreateUserResponse>(
   'UserManagement',
   'createUser',
   request
  );
 }

 /**
  * Update an existing user's information
  */
 static async updateUser(request: UpdateUserRequest): Promise<UpdateUserResponse> {
  if (!request) {
   throw new Error('Request parameter is required');
  }

  return await this.messageRouter.call<UpdateUserResponse>(
   'UserManagement',
   'updateUser',
   request
  );
 }
```

```
}

/**
 * User creation request containing user details
 */
export interface CreateUserRequest {
  /** User's email address - must be unique */
  email: string;

  /** User's full name */
  name: string;

  /** Optional department assignment */
  department?: string;

  /** Extended user profile information */
  profile?: UserProfile;
}

/**
 * Response containing the created user ID and success status
 */
export interface CreateUserResponse {
  /** Unique identifier for the created user */
  userId: string;

  /** Indicates whether the operation succeeded */
  success: boolean;

  /** Complete user object with all populated fields */
  user?: User;
}

/**
 * Extended user profile information
 */
export interface UserProfile {
  /** URL to user's avatar image */
  avatar?: string;

  /** User's biographical information */
  bio?: string;

  /** User's geographic location */
  location?: string;
```

```
  /** User's timezone (e.g., America/New_York) */
  timezone?: string;
}
```

Python Code Generation:

python

```python
# Generated from UserManagement.openrpc.json
from dataclasses import dataclass
from typing import Optional
from platform.messaging import MessageRouter

class UserManagement:
    """
    User management service functions
    Generated from UserManagement v1.2.4
    """

    _message_router = MessageRouter()

    @classmethod
    async def create_user(cls, request: 'CreateUserRequest') -> 'CreateUserResponse':
        """
        Create a new user account

        Args:
            request: User creation request containing user details

        Returns:
            Response containing the created user ID and success status

        Raises:
            ValidationError: When input validation fails
            BusinessLogicError: When business rules are violated
        """
        if request is None:
            raise ValueError("Request parameter is required")

        return await cls._message_router.call(
            service_name="UserManagement",
            method_name="createUser",
            request=request,
            response_type=CreateUserResponse
        )

    @classmethod
    async def update_user(cls, request: 'UpdateUserRequest') -> 'UpdateUserResponse':
        """Update an existing user's information"""
        if request is None:
            raise ValueError("Request parameter is required")
```

```
        return await cls._message_router.call(
            service_name="UserManagement",
            method_name="updateUser",
            request=request,
            response_type=UpdateUserResponse
        )

@dataclass
class CreateUserRequest:
    """User creation request containing user details"""

    email: str # User's email address - must be unique
    name: str  # User's full name
    department: Optional[str] = None # Optional department assignment
    profile: Optional['UserProfile'] = None # Extended user profile information

@dataclass
class CreateUserResponse:
    """Response containing the created user ID and success status"""

    user_id: str    # Unique identifier for the created user
    success: bool   # Indicates whether the operation succeeded
    user: Optional['User'] = None # Complete user object with all populated fields

@dataclass
class UserProfile:
    """Extended user profile information"""

    avatar: Optional[str] = None  # URL to user's avatar image
    bio: Optional[str] = None     # User's biographical information
    location: Optional[str] = None # User's geographic location
    timezone: Optional[str] = None # User's timezone (e.g., America/New_York)
```

Build Pipeline Integration

Code Generation Process:

yaml

```yaml
# .github/workflows/generate-clients.yml
name: Generate Client Libraries
on:
 push:
  paths:
   - 'services/**/*.openrpc.json'
   - 'schemas/**/*.json'

jobs:
 generate-clients:
  runs-on: ubuntu-latest
  steps:
   - uses: actions/checkout@v3

   - name: Install OpenRPC Generator
     run: npm install -g @open-rpc/generator

   - name: Generate C# Client
     run: |
       openrpc-generator \
        --input services/UserManagement.openrpc.json \
        --output generated/csharp \
        --template csharp-functions

   - name: Generate TypeScript Client
     run: |
       openrpc-generator \
        --input services/UserManagement.openrpc.json \
        --output generated/typescript \
        --template typescript-functions

   - name: Generate Python Client
     run: |
       openrpc-generator \
        --input services/UserManagement.openrpc.json \
        --output generated/python \
        --template python-functions

   - name: Commit Generated Code
     run: |
       git add generated/
       git commit -m "Auto-generate client libraries"
       git push
```

Unified Type System

Database Schema Integration

The same JSON schemas used for service functions also define database collection schemas, ensuring complete consistency between service interfaces and data storage.

Database Collection Schema:

json

```
{
 "database": "application",
 "collections": {
  "users": {
   "schema": {
    "$ref": "common-schemas.json#/components/schemas/User"
   },
   "indexes": [
    {
     "fields": ["email"],
     "unique": true
    },
    {
     "fields": ["department", "createdAt"]
    }
   ],
   "validation": {
    "level": "strict",
    "action": "error"
   }
  },
  "orders": {
   "schema": {
    "$ref": "common-schemas.json#/components/schemas/Order"
   },
   "indexes": [
    {
     "fields": ["userId", "createdAt"]
    },
    {
     "fields": ["status"]
    }
   ]
  }
 }
}
```

UI Model Integration

UI components reference the same schema definitions, ensuring that user interfaces remain synchronized with service function signatures and database schemas.

UI Model Definition:

json

```
{
 "ui_models": {
  "UserFormModel": {
   "extends": {
    "$ref": "common-schemas.json#/components/schemas/CreateUserRequest"
   },
   "ui_hints": {
    "email": {
     "component": "email_input",
     "placeholder": "Enter email address",
     "validation_message": "Please enter a valid email address"
    },
    "name": {
     "component": "text_input",
     "placeholder": "Enter full name",
     "validation_message": "Name is required"
    },
    "department": {
     "component": "select",
     "options_source": "DepartmentService.listDepartments",
     "placeholder": "Select department"
    }
   }
  },
  "UserDisplayModel": {
   "extends": {
    "$ref": "common-schemas.json#/components/schemas/User"
   },
   "ui_hints": {
    "avatar": {
     "component": "avatar_image",
     "size": "medium"
    },
    "email": {
     "component": "readonly_text",
     "mask": "email"
    }
   }
  }
 }
}
```

Schema Evolution Coordination

When schemas evolve, the unified type system ensures that changes are coordinated across all platform components.

Non-Breaking Changes (Minor Version):

```json
{
 "migration_type": "compatible",
 "changes": [
  {
   "type": "add_optional_parameter",
   "method": "createUser",
   "parameter": {
    "name": "phoneNumber",
    "schema": {"type": "string", "pattern": "^\\+?[1-9]\\d{1,14}$"},
    "required": false
   }
  },
  {
   "type": "add_optional_response_field",
   "method": "createUser",
   "field": {
    "name": "verificationToken",
    "schema": {"type": "string"}
   }
  }
 ],
 "deployment_strategy": "rolling_update",
 "client_impact": "none"
}
```

Breaking Changes (Major Version):

json

```
{
 "migration_type": "breaking",
 "changes": [
  {
   "type": "rename_parameter",
   "method": "createUser",
   "old_name": "fullName",
   "new_name": "name"
  },
  {
   "type": "change_parameter_type",
   "method": "updateUser",
   "parameter": "userId",
   "old_type": {"type": "string"},
   "new_type": {"type": "string", "format": "uuid"}
  }
 ],
 "deployment_strategy": "blue_green",
 "client_impact": "requires_update",
 "migration_deadline": "2025-06-01"
}
```

Automated Compatibility Checking

CI/CD Pipeline Integration:

yaml

```yaml
# .github/workflows/schema-compatibility.yml
name: Schema Compatibility Check
on:
 pull_request:
  paths:
   - 'services/**/*.openrpc.json'
   - 'schemas/**/*.json'

jobs:
 compatibility-check:
  runs-on: ubuntu-latest
  steps:
   - uses: actions/checkout@v3
     with:
      fetch-depth: 0 # Need full history for comparison

   - name: Get Changed Schemas
     id: changed-files
     run: |
       echo "schemas=$(git diff --name-only HEAD~1 HEAD | grep -E '\.(openrpc\.json|schema\.json) | tr '\n' ' ')" >> $

   - name: Check Schema Compatibility
     run: |
       for schema in ${{ steps.changed-files.outputs.schemas }}; do
         echo "Checking compatibility for $schema"
         schema-compatibility-checker \
           --current "origin/main:$schema" \
           --proposed "$schema" \
           --output-format json \
           --fail-on-breaking
       done

   - name: Update Migration Plans
     if: contains(steps.changed-files.outputs.schemas, '.openrpc.json')
     run: |
       migration-planner \
         --input ${{ steps.changed-files.outputs.schemas }} \
         --output migration-plans/ \
         --check-all-callers
```

Runtime Compatibility Validation:

csharp

```csharp
public class RuntimeCompatibilityValidator
{
    public async Task<bool> ValidateServiceCompatibility(string serviceName, string version)
    {
        var callerRequirements = await LoadCallerRequirements(serviceName);
        var serviceCapabilities = await LoadServiceCapabilities(serviceName, version);

        foreach (var requirement in callerRequirements)
        {
            var isCompatible = CheckMethodCompatibility(requirement, serviceCapabilities);
            if (!isCompatible)
            {
                LogCompatibilityError(serviceName, requirement.MethodName, version);
                return false;
            }
        }

        return true;
    }
}
```

Security and Validation

Input Validation

All service function inputs are automatically validated against their JSON Schema definitions before execution.

Validation Pipeline:

csharp

```csharp
public class SchemaValidationMiddleware
{
    public async Task InvokeAsync(MessageContext context, RequestDelegate next)
    {
        var method = ResolveMethod(context.ServiceName, context.MethodName);
        var parameterSchema = method.Parameters[0].Schema;

        var validationResult = ValidateInput(context.RequestBody, parameterSchema);
        if (!validationResult.IsValid)
        {
            context.Response = CreateValidationErrorResponse(validationResult.Errors);
            return;
        }

        await next(context);
    }

    private ValidationResult ValidateInput(object input, JsonSchema schema)
    {
        var validator = new JsonSchemaValidator();
        return validator.Validate(input, schema);
    }
}
```

Schema Integrity: OpenRPC specifications and JSON schemas are cryptographically signed to ensure they haven't been tampered with during deployment or storage.

csharp

```csharp
public class SchemaSecurityValidator
{
  private readonly ICryptographicService _crypto;

  public async Task<bool> ValidateSchemaIntegrity(OpenRpcSpec spec, string signature)
  {
    var specJson = JsonSerializer.Serialize(spec);
    var isValid = await _crypto.VerifySignature(specJson, signature);

    if (!isValid)
    {
      throw new SecurityException("Schema signature validation failed");
    }

    return true;
  }

  public ValidationResult ValidateInput(object input, JsonSchema schema, UserContext userContext)
  {
    var result = new ValidationResult();

    // Basic schema validation
    var schemaValidation = ValidateAgainstSchema(input, schema);
    if (!schemaValidation.IsValid)
    {
      result.AddErrors(schemaValidation.Errors);
      return result;
    }

    // Security-specific validation
    ValidateForSecurityThreats(input, result);
    ValidateUserPermissions(input, schema, userContext, result);

    return result;
  }

  private void ValidateForSecurityThreats(object input, ValidationResult result)
  {
    // Check for common injection attacks
    // Validate string lengths to prevent DoS
    // Check for suspicious patterns
  }
}
```

Implementation Guidelines

Service Function Development

Creating New Service Functions:

1. **Define OpenRPC Specification**: Start with a complete OpenRPC document describing all methods
2. **Validate Schemas**: Ensure all JSON schemas are valid and well-documented
3. **Generate Code Stubs**: Use code generation tools to create function stubs in target languages
4. **Implement Business Logic**: Fill in the generated stubs with actual business logic
5. **Test Compatibility**: Verify that the implementation matches the OpenRPC specification

Example Development Workflow:

```bash
# 1. Create OpenRPC specification
nano services/UserManagement.openrpc.json

# 2. Validate specification
openrpc-validator services/UserManagement.openrpc.json

# 3. Generate implementation stubs
openrpc-generator \
 --input services/UserManagement.openrpc.json \
 --output src/UserManagement \
 --template csharp-service-stub

# 4. Implement business logic
# Edit generated files to add actual implementation

# 5. Test against specification
platform-test-runner \
 --service UserManagement \
 --spec services/UserManagement.openrpc.json
```

Schema Design Best Practices

Type Definition Guidelines:

1. **Use Descriptive Names**: Schema names should clearly indicate their purpose
2. **Provide Documentation**: Include descriptions for all schemas, properties, and methods
3. **Define Constraints**: Use JSON Schema validation rules (minLength, maxLength, pattern, etc.)
4. **Leverage Composition**: Use `allOf`, `oneOf`, and `$ref` for schema reuse
5. **Version Schemas**: Include version information in schema metadata

Schema Organization:

```
schemas/
├── common/
│   ├── base-types.json        # Fundamental types (id, timestamp, etc.)
│   ├── error-types.json       # Standard error schemas
│   └── pagination.json        # Pagination and filtering schemas
├── entities/
│   ├── user.json              # User entity and related types
│   ├── order.json             # Order entity and related types
│   └── product.json           # Product entity and related types
└── services/
    ├── UserManagement.openrpc.json
    ├── OrderManagement.openrpc.json
    └── ProductCatalog.openrpc.json
```

Validation and Testing

Schema Validation Process:

csharp

```csharp
public class SchemaValidator
{
    public ValidationResult ValidateOpenRpcSpec(string specPath)
    {
        var spec = LoadOpenRpcSpec(specPath);
        var result = new ValidationResult();

        // Validate OpenRPC structure
        if (!IsValidOpenRpcVersion(spec.OpenRpc))
        {
            result.AddError("Invalid OpenRPC version");
        }

        // Validate all referenced schemas
        foreach (var method in spec.Methods)
        {
            ValidateMethodSchemas(method, result);
        }

        // Check for schema consistency
        ValidateSchemaReferences(spec, result);

        return result;
    }

    private void ValidateMethodSchemas(Method method, ValidationResult result)
    {
        // Validate parameter schemas
        foreach (var param in method.Params)
        {
            if (!IsValidJsonSchema(param.Schema))
            {
                result.AddError($"Invalid schema for parameter {param.Name}");
            }
        }

        // Validate result schema
        if (!IsValidJsonSchema(method.Result.Schema))
        {
            result.AddError($"Invalid result schema for method {method.Name}");
        }
    }
}
```

Integration Testing Framework:

```csharp
[TestClass]
public class ServiceCompatibilityTests
{
  [TestMethod]
  public void UserManagement_CreateUser_MatchesSpecification()
  {
    // Load OpenRPC specification
    var spec = LoadOpenRpcSpec("UserManagement.openrpc.json");
    var createUserMethod = spec.Methods.First(m => m.Name == "createUser");

    // Create test request matching parameter schema
    var testRequest = CreateValidRequest(createUserMethod.Params[0].Schema);

    // Call actual service function
    var response = UserManagement.CreateUser(testRequest).Result;

    // Validate response matches result schema
    var isValidResponse = ValidateAgainstSchema(response, createUserMethod.Result.Schema);
    Assert.IsTrue(isValidResponse, "Service response does not match OpenRPC specification");
  }

  [TestMethod]
  public void AllServices_MatchTheirSpecifications()
  {
    var serviceSpecs = LoadAllServiceSpecs();

    foreach (var spec in serviceSpecs)
    {
      ValidateServiceImplementation(spec);
    }
  }
}
```

Coordinated Schema Updates

Multi-Component Update Process

When schemas change, the platform coordinates updates across all affected components to maintain consistency.

Coordinated Schema Update Example:

json


```
{
 "schema_migration": {
  "version": "1.2.4_to_1.2.5",
  "changes": [
   {
    "type": "add_optional_field",
    "schema": "User",
    "field": "phoneNumber",
    "field_schema": {
     "type": "string",
     "pattern": "^\\+?[1-9]\\d{1,14}$",
     "description": "User's phone number in E.164 format"
    }
   }
  ],
  "affected_components": {
   "services": [
    {
     "name": "UserManagement",
     "methods": ["createUser", "updateUser"],
     "compatibility": "backward_compatible"
    }
   ],
   "database": [
    {
     "collection": "users",
     "migration": "add_optional_column",
     "compatibility": "fully_compatible"
    }
   ],
   "ui": [
    {
     "model": "UserFormModel",
     "changes": ["add_optional_phone_field"],
     "compatibility": "backward_compatible"
    }
   ]
  },
  "deployment_strategy": "rolling_update",
  "rollback_plan": "remove_optional_field"
 }
}
```

Migration Automation

Database Migration Generation:

csharp

```csharp
public class DatabaseMigrationGenerator
{
  public Migration GenerateMigration(SchemaMigration schemaMigration)
  {
    var migration = new Migration
    {
      Version = schemaMigration.Version,
      Description = $"Schema migration {schemaMigration.Version}"
    };

    foreach (var change in schemaMigration.Changes)
    {
      switch (change.Type)
      {
        case "add_optional_field":
          migration.AddOperation(new AddColumnOperation
          {
            Table = GetTableForSchema(change.Schema),
            Column = new ColumnDefinition
            {
              Name = change.Field,
              Type = MapSchemaTypeToSqlType(change.FieldSchema),
              IsNullable = true
            }
          });
          break;

        case "remove_field":
          migration.AddOperation(new DropColumnOperation
          {
            Table = GetTableForSchema(change.Schema),
            Column = change.Field
          });
          break;

        case "change_field_type":
          migration.AddOperation(new AlterColumnOperation
          {
            Table = GetTableForSchema(change.Schema),
            Column = change.Field,
            NewType = MapSchemaTypeToSqlType(change.NewFieldSchema)
          });
          break;
```

```
        }
      }

      return migration;
    }
  }
```

UI Model Update Generation:

typescript

```
export class UIModelGenerator {
 generateUpdatedModel(migration: SchemaMigration): UIModelUpdate {
  const updates: UIModelUpdate = {
   version: migration.version,
   models: []
  };

  for (const change of migration.changes) {
   if (change.type === 'add_optional_field') {
    updates.models.push({
     name: `${change.schema}FormModel`,
     changes: [{
      type: 'add_field',
      field: change.field,
      component: this.inferUIComponent(change.fieldSchema),
      validation: this.generateValidationRules(change.fieldSchema),
      required: false
     }]
    });
   }
  }

  return updates;
 }

 private inferUIComponent(schema: JsonSchema): string {
  if (schema.format === 'email') return 'email_input';
  if (schema.format === 'date') return 'date_picker';
  if (schema.pattern?.includes('phone')) return 'phone_input';
  if (schema.enum) return 'select';
  return 'text_input';
 }
}
```

Conclusion

The Schema and API Specification system provides a comprehensive foundation for the Service Function Architecture platform through the integration of OpenRPC and JSON Schema standards. This unified approach ensures type safety, compatibility management, and automated code generation across all platform components.

Key Benefits Delivered:

- **Complete Type Safety**: All service functions, database schemas, and UI models share consistent type definitions
- **Automated Compatibility**: Schema evolution is managed through automated compatibility checking and migration planning
- **Cross-Language Support**: Native code generation provides type-safe interfaces in multiple programming languages
- **Unified Development**: Single specification format covers service functions, data storage, and user interface models
- **Enterprise Security**: Input validation, schema integrity, and security validation are built into the specification system

Strategic Advantages:

- **Reduced Development Time**: Automatic code generation eliminates boilerplate and ensures consistency
- **Improved Reliability**: Type safety and validation prevent runtime errors and data inconsistencies
- **Simplified Maintenance**: Coordinated schema evolution ensures all components remain synchronized
- **Enhanced Developer Experience**: Rich tooling support and IDE integration accelerate development workflows
- **Future-Proof Architecture**: Standards-based approach enables long-term evolution and ecosystem growth

The OpenRPC and JSON Schema foundation provides the platform with enterprise-grade type safety and compatibility management while maintaining the flexibility needed for rapid development and deployment in modern distributed systems.