Aggregate CRD Design Options

OpenAPI Operator Generator

This document outlines design options for adding an aggregate CRD feature to the OpenAPI operator generator. An aggregate CRD takes one or more generated CRDs as input and combines them for unified management or observation.

Option 1: Composition CRD (Reference-Based)

A CRD that references existing Custom Resources by name/namespace and aggregates their statuses.

```
apiVersion: petstore.example.com/v1alpha1
kind: PetStoreAggregate
metadata:
  name: my-store-health
spec:
  resources:
    - kind: Pet
      name: my-pet
      namespace: default
    - kind: Order
      name: my-order
      namespace: default
  aggregationStrategy: AllHealthy  # or AnyHealthy, Quorum
status:
  state: Synced  # Aggregated state
  resourceStatuses:
    - kind: Pet
      name: my-pet
      state: Synced
      externalID: "123"
    - kind: Order
      name: my-order
      state: Failed
      message: "API returned 404"
  conditions: [...]
```

Pros:

- Clean separation - resources managed independently
- Leverages existing AllHealthy pattern in pkg/endpoint/resolver.go
- Easy to understand ownership model

Cons:

- Requires watching multiple resource types
- Resources must exist before aggregate

Option 2: Inline Composition CRD (Embedded Specs)

An aggregate CRD that embeds the specs of child resources directly, creating multiple API resources from a single CR.

```
apiVersion: petstore.example.com/v1alpha1
kind: PetBundle
metadata:
  name: full-pet-setup
spec:
  pet:
    name: "Fido"
    status: available
  order:
    petId: "{{.pet.externalID}}"  # Template reference
    quantity: 1
status:
  pet:
    state: Synced
    externalID: "123"
  order:
    state: Synced
    externalID: "456"
```

Pros:

- Single CR creates multiple API resources
- Can express dependencies between resources (order references pet ID)
- Familiar pattern (similar to Helm charts, Argo Application)

Cons:

- More complex reconciliation logic (ordering, dependency resolution)
- Harder to manage partial failures
- Spec becomes large

Option 3: Declarative Bundle (Template-Based)

A meta-CRD that declares a bundle template, instantiated by a separate CR. Provides reusable templates with parameters.

```
# Template definition
apiVersion: petstore.example.com/v1alpha1
kind: PetBundleTemplate
metadata:
  name: pet-with-order
spec:
  parameters:
    - name: petName
      type: string
    - name: quantity
      type: integer
      default: 1
  resources:
    - apiVersion: petstore.example.com/v1alpha1
      kind: Pet
      spec:
        name: "{{ .petName }}"
    - apiVersion: petstore.example.com/v1alpha1
      kind: Order
      spec:
        petId: "{{ .resources.Pet.status.externalID }}"
        quantity: "{{ .quantity }}"
---
# Instance
apiVersion: petstore.example.com/v1alpha1
kind: PetBundle
metadata:
  name: my-bundle
spec:
  templateRef: pet-with-order
  parameters:
    petName: "Buddy"
    quantity: 2
```

Pros:

- Reusable templates
- Clear separation of template definition and instantiation
- Parameters enable customization

Cons:

- Most complex to implement
- Two new CRD types needed
- Template language adds complexity

Option 4: Status Aggregator (Read-Only)

A lightweight CRD that only observes and aggregates status from existing resources. Similar to the existing ReadOnly mode.

```yaml
apiVersion: petstore.example.com/v1alpha1
kind: ResourceHealthCheck
metadata:
  name: store-health
spec:
  selector:
    matchLabels:
      app: petstore
  # OR explicit list
  resources:
    - group: petstore.example.com
      kind: Pet
    - group: petstore.example.com
      kind: Order
  aggregation:
    type: Summary  # or Detailed
status:
  summary:
    total: 10
    synced: 8
    failed: 2
  conditions:
    - type: AllHealthy
      status: "False"
      reason: "2 resources failed"
```

Pros:

- Simplest to implement - read-only aggregation
- Builds on existing ReadOnly mode pattern
- Good for dashboards/monitoring

Cons:

- Cannot create/manage resources
- Limited to status observation

Option 5: Workflow/Pipeline CRD

An ordered sequence of operations with dependencies. Steps execute in order based on dependency graph.

```
apiVersion: petstore.example.com/v1alpha1
kind: PetWorkflow
metadata:
  name: onboard-pet
spec:
  steps:
    - name: create-pet
      resource:
        kind: Pet
        spec:
          name: "Fido"
          status: pending
    - name: upload-image
      dependsOn: [create-pet]
      action:
        kind: PetUploadImage
        spec:
          petId: "{{ steps.create-pet.externalID }}"
          file: "..."
    - name: mark-available
      dependsOn: [upload-image]
      resource:
        kind: Pet
        spec:
          name: "Fido"
          status: available
status:
  phase: Running  # Pending, Running, Succeeded, Failed
  steps:
    create-pet: Completed
    upload-image: Running
    mark-available: Pending
```

Pros:

- Expresses ordered operations naturally
- Good fit for Action endpoints (one-shot operations)
- Clear execution semantics

Cons:

- Most complex reconciliation logic
- Need DAG evaluation
- State machine complexity

Implementation Considerations

Aspect	Current Support	What's Needed
Multi-resource status	Yes (Responses map)	Extend to cross-CRD

Type references	Yes (UsesSharedType)	Cross-CRD references
Health aggregation	Yes (AllHealthy strategy)	Apply to CRs not endpoints
Watches	Single type	Multi-type watches
Templates	Go templates	Add aggregate templates

Recommendation

For a first iteration, **Option 1 (Composition CRD)** or **Option 4 (Status Aggregator)** are recommended:

- **Option 1** - If you need the aggregate to manage lifecycle (create/delete child resources)
- **Option 4** - If you only need observability across resources

Both leverage existing patterns (Responses map, AllHealthy strategy, ReadOnly mode) and require minimal new concepts.

Crossplane function-status-transformer Analysis

Overview

The Crossplane **function-status-transformer** is a composition function that provides a declarative, matcher-based approach to status aggregation. This pattern is directly applicable to the aggregate CRD design.

What It Does:

- Watches composed resources for status condition changes
- Matches conditions using configurable matchers (exact match, regex, wildcards)
- Transforms and propagates matched conditions to parent composite resource
- Emits Kubernetes events based on condition transitions

Key Design Pattern: Matcher-to-Action Pipeline

The function implements a hook-based pattern where matchers select resources and conditions, then actions propagate transformed status to the parent.

```
StatusConditionHooks[]
  |-- Matchers (AND logic between matchers)
  |     |-- resources: ["cloudsql-\\d+"]    # regex selection
  |     |-- conditions:                      # what to match
  |     |       type: Synced
  |     |       status: "False"
  |     |       message: "error: (?P<Error>.+)"  # capture groups
  |     +-- matchType: AnyResourceMatchesAllConditions
  +-- Actions
        |-- setConditions:                   # propagate to parent
        |       type: DatabaseReady
        |       message: "Failed: {{ .Error }}"  # template injection
        +-- createEvents: [...]
```

Match Types

- **AnyResourceMatchesAnyCondition** - Any resource matches any condition
- **AnyResourceMatchesAllConditions** - Any resource matches all conditions
- **AllResourcesMatchAnyCondition** - All resources match any condition
- **AllResourcesMatchAllConditions** - All resources match all conditions (default)

Key Features

- **Regex capture groups** - Extract error details: message: "error: (?P<Error>.+)"
- **Template injection** - Use captured data: "Failed: {{ .Error }}"
- **Sequential processing** - Hooks execute in order, later hooks see earlier state
- **Non-blocking errors** - Failures set condition status, don't halt reconciliation

Application to Aggregate CRD

The function-status-transformer pattern maps well to **Option 4 (Status Aggregator)** with potential extensions to **Option 1 (Composition CRD)**.

```
apiVersion: petstore.example.com/v1alpha1
kind: ResourceAggregate
metadata:
  name: pet-health
spec:
  # Resource selection (similar to Crossplane matchers)
  resourceSelectors:
    - group: petstore.example.com
      kind: Pet
      matchLabels:
        app: petstore
    - group: petstore.example.com
      kind: Order
      namePattern: "order-.*"  # regex support

  # Status condition hooks (from function-status-transformer)
  statusConditionHooks:
    - matchers:
        - resources:
            kind: Pet
          conditions:
            - type: Synced
              status: "False"
              reason: "APIError"
              message: "(?P<Error>.+)"
        - matchType: AnyResourceMatchesAnyCondition
      setConditions:
        - type: PetsHealthy
          status: "False"
          reason: SyncFailure
          message: "Pet sync failed: {{ .Error }}"

    - matchers:
        - resources:
            kind: "*"  # all resources
          conditions:
            - type: Synced
              status: "True"
        - matchType: AllResourcesMatchAllConditions
      setConditions:
        - type: Ready
          status: "True"
          reason: AllSynced
          message: "All {{ .MatchedCount }} resources synced"

status:
  conditions:
    - type: Ready
      status: "False"
      reason: SyncFailure
    - type: PetsHealthy
      status: "False"
      message: "Pet sync failed: API returned 503"
  matchedResources:
    - kind: Pet
      name: my-pet
      state: Failed
    - kind: Order
      name: order-123
```

```
state: Synced
```

Pattern Mapping

Crossplane Pattern	Application to Generator
Matcher-based selection	Select resources by kind, name regex, labels
Condition matching	Match on State, ExternalID, Message fields
Regex capture groups	Extract error details for aggregated messages
Match types (Any/All)	AllHealthy, AnyHealthy, Quorum(n) strategies
Template injection	{{ .Error }}, {{ .MatchedCount }} in messages
Non-blocking errors	Set condition to False rather than failing reconciliation
Sequential hooks	Process hooks in order, later hooks see earlier state

Differences from Crossplane

Aspect	Crossplane	OpenAPI Generator
Resource type	Generic K8s resources	Generated CRDs only
Execution model	gRPC function pipeline	In-controller reconciliation
Status source	status.conditions[]	status.state, status.message
Discovery	Composition defines resources	Selector-based discovery

Implementation Sketch

```
// In pkg/mapper/resource.go - new aggregate type
type AggregateCRDDefinition struct {
    Kind              string
    ResourceSelectors []ResourceSelector
    StatusHooks       []StatusConditionHook
}

type ResourceSelector struct {
```

```
    Group       string
    Kind        string
    NamePattern string            // regex
    MatchLabels map[string]string
}

type StatusConditionHook struct {
    Matchers      []ConditionMatcher
    MatchType     MatchType  // AnyAny, AnyAll, AllAny, AllAll
    SetConditions []ConditionTemplate
}

type ConditionMatcher struct {
    ResourceKind string
    Conditions   []ConditionMatch
}

type ConditionMatch struct {
    Field   string // "State", "Message", "ExternalID"
    Pattern string // regex with capture groups
}
```

Crossplane Conclusion

The function-status-transformer design is a **strong fit** for the aggregate CRD feature. Recommendations:

1. **Adopt the matcher-to-action pattern** - It's declarative, flexible, and battle-tested
2. **Simplify for your use case** - No gRPC pipelines needed; inline logic in controller
3. **Use existing status fields** - Match on State, Message, ExternalID instead of conditions
4. **Add match types** - AllHealthy, AnyHealthy, Quorum(n) map to Crossplane's match types

Sources:

- github.com/crossplane-contrib/function-status-transformer
- github.com/crossplane/crossplane/blob/main/design/design-doc-composition-functions.md

Kro (Kube Resource Orchestrator) Analysis

Overview

Kro (Kube Resource Orchestrator) is a Kubernetes-native framework from AWS, Google, and Microsoft that enables defining groups of resources as a single custom resource. It's now a Kubernetes SIG Cloud Provider subproject.

Key Characteristics:

- Kubernetes-native, cloud-agnostic resource composition
- Dynamic CRD generation from ResourceGraphDefinition
- CEL (Common Expression Language) for data flow between resources
- Automatic DAG-based dependency detection
- Conditional resource creation with includeWhen

Core Design: ResourceGraphDefinition

Kro's central concept is the **ResourceGraphDefinition** (RGD), which defines a user-facing schema and the underlying resources to create.

```
apiVersion: kro.run/v1alpha1
kind: ResourceGraphDefinition
metadata:
  name: web-application
spec:
  # Schema: defines the user-facing API
  schema:
    apiVersion: v1alpha1
    kind: WebApplication
    spec:
      name: string | required=true immutable=true
      replicas: integer | default=1 minimum=1 maximum=100
      image: string | required=true
    status:
      availableReplicas: ${deployment.status.availableReplicas}
      serviceEndpoint: ${service.status.loadBalancer.ingress[0].hostname}
      allReady: ${deployment.status.availableReplicas == schema.spec.replicas}

  # Resources: the underlying resources to create
  resources:
    - id: deployment
      template:
        apiVersion: apps/v1
        kind: Deployment
```

```
    metadata:
      name: ${schema.spec.name}
    spec:
      replicas: ${schema.spec.replicas}
      template:
        spec:
          containers:
            - name: app
              image: ${schema.spec.image}
  readyWhen:
    - ${deployment.status.availableReplicas == schema.spec.replicas}

- id: service
  template:
    apiVersion: v1
    kind: Service
    metadata:
      name: ${schema.spec.name}-svc
  includeWhen:
    - ${schema.spec.exposeService == true}
```

Key Design Patterns

Pattern	Description
Dynamic CRD Generation	RGD creates a new CRD (e.g., WebApplication) at runtime
CEL Expressions	${resource.field} syntax for data flow between resources
DAG Dependencies	Auto-detected from CEL references; validated for cycles
readyWhen	CEL conditions that determine when a resource is ready
includeWhen	CEL conditions for conditional resource creation
Status Propagation	CEL expressions in schema.status pull from child resources

How Dependencies Work

Kro automatically determines dependencies by analyzing CEL expressions. References like **${database.status.endpoint}** create implicit dependencies.

```
resources:
  - id: database
    template:
      kind: RDSInstance
      spec:
        name: ${schema.spec.dbName}

  - id: secret
    template:
      kind: Secret
      data:
        # References database - creates implicit dependency
        connectionString: ${database.status.endpoint}

  - id: deployment
    template:
      kind: Deployment
      spec:
        env:
          # References secret - creates implicit dependency
          - name: DB_URL
            valueFrom:
              secretKeyRef:
                name: ${secret.metadata.name}

# Kro builds DAG: database -> secret -> deployment
```

Status Aggregation Pattern

```
spec:
  schema:
    status:
      # Direct propagation
      dbEndpoint: ${database.status.endpoint}

      # Computed aggregation
      allReady: ${
        deployment.status.conditions.exists(c,
          c.type == 'Available' && c.status == 'True') &&
        database.status.conditions.exists(c,
          c.type == 'Ready' && c.status == 'True')
      }

      # Count aggregation
      totalReplicas: ${deployment.status.replicas + statefulset.status.replicas}

      # Optional access (won't fail if field missing)
      endpoint: ${service.status.?loadBalancer.?ingress[0].hostname}
```

Application to Aggregate CRD

The Kro pattern maps well to **Option 2 (Inline Composition)** and **Option 3 (Template-Based)** from the design options.

```
apiVersion: petstore.example.com/v1alpha1
kind: PetBundleDefinition  # Like ResourceGraphDefinition
metadata:
  name: pet-with-order
spec:
  # User-facing schema (generates CRD: PetBundle)
  schema:
    apiVersion: v1alpha1
    kind: PetBundle
    spec:
      petName: string | required=true
      petStatus: string | default="available"
      orderQuantity: integer | default=1 minimum=1
    status:
      petId: ${pet.status.externalID}
      orderId: ${order.status.externalID}
      allSynced: ${pet.status.state == "Synced" &&
                  order.status.state == "Synced"}

  # Resources to create (your generated CRDs)
  resources:
    - id: pet
      template:
        apiVersion: petstore.example.com/v1alpha1
        kind: Pet
        spec:
          name: ${schema.spec.petName}
          status: ${schema.spec.petStatus}
      readyWhen:
        - ${pet.status.state == "Synced"}

    - id: order
      template:
        apiVersion: petstore.example.com/v1alpha1
        kind: Order
        spec:
          petId: ${pet.status.externalID}  # Dependency
          quantity: ${schema.spec.orderQuantity}
      readyWhen:
        - ${order.status.state == "Synced"}
      includeWhen:
        - ${schema.spec.orderQuantity > 0}
```

Pattern Mapping

Kro Pattern	Application to Your Generator
ResourceGraphDefinition	Aggregate CRD definition (compile-time or runtime)
CEL expressions	Reference fields between generated CRDs
Auto DAG detection	Parse ${resource.field} to build dependency graph
readyWhen	${pet.status.state == "Synced"}
includeWhen	Conditional resource creation
Status propagation	status.allSynced: ${pet.status.state == "Synced" && ...}
Dynamic CRD	Generate aggregate CRD from definition

Key Differences from Kro

Aspect	Kro	Your Generator
When CRDs created	Runtime (dynamic)	Compile-time (code generation)
Resource types	Any K8s resource	Your generated CRDs only
Controller	Dynamic microcontroller per RGD	Generated controller code
Expression language	CEL	Go templates or CEL
Scope	General-purpose	OpenAPI-specific

Implementation Options

Option A: Compile-Time Generation (Simpler)

```
// In pkg/mapper/resource.go
type BundleDefinition struct {
    Kind      string
    Schema    BundleSchema
```

```
    Resources []BundleResource
}

type BundleResource struct {
    ID          string
    Kind        string            // Reference to generated CRD
    SpecMap     map[string]string // Field -> CEL expression
    ReadyWhen   []string          // CEL conditions
    IncludeWhen []string          // CEL conditions
}

// Generate a bundle controller that:
// 1. Parses CEL expressions to find dependencies
// 2. Creates resources in DAG order
// 3. Evaluates readyWhen/includeWhen conditions
// 4. Propagates status via CEL expressions
```

Option B: Runtime Definition (Like Kro)

Add a BundleDefinition CRD that users apply at runtime. The controller watches BundleDefinition resources, dynamically creates CRDs, and spawns reconcilers for each bundle type.

Comparison: Kro vs Crossplane function-status-transformer

Aspect	Kro	function-status-transformer
Primary purpose	Resource composition	Status aggregation
Creates resources	Yes (full lifecycle)	No (read-only)
Dependencies	DAG from CEL references	Not applicable
Conditional creation	Yes (includeWhen)	No
Status propagation	CEL in schema.status	Matcher -> setConditions
Expression language	CEL	CEL + regex capture
Complexity	Higher	Lower

Kro Conclusion

Kro's ResourceGraphDefinition pattern is a **strong fit** for implementing Option 2 (Inline Composition) or Option 3 (Template-Based) aggregate CRDs.

Recommendations:

- **Use Kro patterns** for Option 2/3 where you need to create/manage multiple resources with dependencies
- **Use function-status-transformer patterns** for Option 1/4 where resources are managed separately
- **Consider CEL** for expression language - it's type-safe and used by both Kro and Crossplane
- **Auto-detect dependencies** from CEL expressions to build DAG

Sources:

- kro.run - Official Documentation
- github.com/awslabs/kro (now kubernetes-sigs/kro)
- AWS Blog - Introducing kro
- Google Cloud Blog - Introducing Kube Resource Orchestrator

Summary: Design Pattern Selection

Based on analysis of Crossplane function-status-transformer and Kro, here is guidance on which patterns to apply for each design option:

Design Option	Recommended Pattern	Key Features to Adopt
Option 1: Composition CRD (Reference-Based)	Crossplane function-status-transformer	Matcher-to-action hooks Regex capture groups Match types (Any/All)
Option 2: Inline Composition (Embedded Specs)	Kro ResourceGraphDefinition	CEL expressions Auto DAG detection readyWhen conditions
Option 3: Declarative Bundle (Template-Based)	Kro ResourceGraphDefinition	Schema definition includeWhen conditionals Status propagation
Option 4: Status Aggregator (Read-Only)	Crossplane function-status-transformer	Selector-based discovery Condition matching Template injection
Option 5: Workflow/Pipeline	Kro + Custom	DAG execution order Step dependencies Phase tracking

Final Recommendation

For the OpenAPI Operator Generator, a hybrid approach is recommended:

1. **Start with Option 4 (Status Aggregator)** using Crossplane's matcher-to-action pattern - lowest complexity, highest value for observability
2. **Extend to Option 1 (Composition CRD)** for lifecycle management of referenced resources
3. **Consider Option 2/3 (Inline/Template)** using Kro patterns if users need single-CR deployment of multiple resources

Both Crossplane and Kro use **CEL** as their expression language. Adopting CEL provides type-safety, validation at creation time, and alignment with the broader Kubernetes ecosystem.