Introduction

The Hologram Container Engine and Platform

Hologram is a universal container engine and platform built on a fundamental insight: **all information can be contained through projection rather than classification**.

Traditional systems organize information through type hierarchies, schemas, and rigid classifications imposed at write time. Hologram inverts this model. Resources exist without inherent type or structure in a universal store, identified only by their content. Meaning, structure, and purpose emerge through **projections**—queries that identify, aggregate, and transform resources into containers.

This document introduces the core concepts, architecture, and operating principles of the Hologram Container Engine and Platform.

What Hologram Is

Hologram is:

- **A universal resource store** using content-addressed storage where resources exist as immutable, undifferentiated data identified by cryptographic hash of their content
- **A projection engine** that executes projection definitions to identify, aggregate, and transform resources into containers with meaning and purpose
- **A container platform** providing base container types (components, instances, interfaces, documentation, tests, managers, views) that are themselves defined through projections
- **A self-describing system** where the platform's own definition exists as projections in the store, enabling evolution without external dependencies

Core Innovation: Containment Through Projection

The central innovation is the **projection-emission cycle**:

1. **Projection** identifies and aggregates resources from the store based on queries
2. **Execution** operates on projected resources, producing results
3. **Emission** writes new resources back to the store as immutable, content-addressed data
4. **Storage** persists emitted resources, making them available for future projections

This cycle applies uniformly across the platform:

- Creating a component definition projects artifacts and emits component resources
- Running an instance projects a component definition and emits logs, state, and application data
- Materializing a view projects resources and emits a persistent representation
- Every operation is a projection that emits

The cycle creates a **self-evolving system** where emitted resources become inputs to

future projections, enabling continuous refinement and extension without modifying the engine or platform core.

Key Principles

1. Resources Have No Inherent Type

Resources in the store are undifferentiated data. A JSON document, a binary blob, a text file—all exist as content with a cryptographic identifier. Type, structure, and meaning emerge through projection, not classification.

2. Content Addressing Provides Identity

Resources are identified by the cryptographic hash of their content (Content Identifier, or CID). Identical content always produces the same CID. This provides:

- **Deduplication** - identical content stored once
- **Integrity** - content cannot change without changing CID
- **Immutability** - resources are write-once, enabling reliable referencing
- **Distribution** - content can move between stores while maintaining identity

3. Projections Create Containers

A **projection** is a query-based definition that identifies resources matching certain criteria, aggregates them according to relationship rules, and transforms them into a container. The same resources can participate in multiple projections, appearing in different containers with different purposes.

4. Emissions Produce New Resources

Executing a projection produces results—these results are **emitted** as new resources to the store. Emissions are themselves projectable, creating feedback loops where operations build on previous results.

5. The Platform Is Self-Describing

The projection engine, container types, and platform operations are all defined as projections in the store. The system contains its own definition, enabling introspection, evolution, and extension through the same mechanisms used for application containers.

Architecture Overview

The platform consists of three fundamental layers:

The Container Store

A content-addressed storage system providing four core capabilities:

- **Store**: Write immutable resources identified by content hash
- **Retrieve**: Read resources by CID
- **Reference**: Link resources through CID references

- **Query**: Identify resources matching projection criteria

The store is implementation-independent—filesystem, database, distributed storage, or cloud object storage can provide the backend.

The Projection Engine

Executes projection definitions to create containers:

- Evaluates queries to identify resources in the store
- Follows references to aggregate related resources
- Applies transformations to produce container contents
- Delivers projected containers to requesters

The engine interprets projection definitions (themselves resources in the store) and orchestrates the projection-emission cycle.

The Container Platform

Provides base container types and operations:

- **Component containers** define reusable capabilities
- **Instance containers** execute component definitions
- **Interface containers** specify contracts between components
- **Documentation containers** provide human-readable views
- **Test containers** validate behavior
- **Manager containers** control lifecycle operations
- **View containers** materialize persistent projections

All platform container types are defined through projections, making them extensible and evolvable.

The Projection-Emission Cycle

The fundamental operating principle is the continuous cycle:

Project → Execute → Emit → Store → Project…

Every interaction with Hologram follows this cycle:

- A client submits a projection request
- The engine identifies and aggregates resources
- Execution produces results
- Results are emitted as new resources to the store
- Emitted resources become available for projection

This creates a **generative system** where operations compound—each emission adds to the store, enabling more sophisticated projections that produce richer emissions.

What Hologram Enables

The projection model provides unique capabilities:

Flexible Organization: Resources can participate in multiple containers simultaneously. A documentation resource might appear in a component container, a searchable

documentation view, and a tutorial container—without duplication or synchronization.

Immutable History: All resources are immutable. Changes create new resources with new CIDs. The complete history exists in the store, enabling time-travel queries and audit trails.

Composable Projections: Projections can project other projections. A view might project components, which themselves project interfaces, documentation, and tests. Composition creates powerful abstractions without runtime overhead.

Self-Evolution: The platform definition exists as projections. Improving Hologram means emitting new projection definitions to the store. The system evolves through the same mechanisms used for applications.

Implementation Independence: The projection model is abstract—the engine, store backend, and platform operations can be implemented in any language, on any infrastructure. Portability is inherent.

Reading Guide

This documentation is organized into four parts, each building on the previous:

Part I: Foundation (Documents 1-4) Establishes first principles: the container store, projection-emission cycle, and how resources become containers through projection.

Part II: The Engine (Documents 5-7) Details how the projection engine executes projections, the projection language for defining projections, and how emissions work.

Part III: The Platform (Documents 8-11) Describes platform container types, views, operations, and the client API for interacting with the platform.

Part IV: System Architecture (Documents 12-15) Covers bootstrap architecture, system properties, implementation considerations, and terminology.

Each document builds on concepts from previous documents. Reading sequentially provides the clearest understanding, though individual documents can serve as reference material.

Next Steps

The next document, **The Container Store**, establishes the foundation by describing how content-addressed storage provides the universal resource container underlying all projections.

From that foundation, we build up the projection-emission cycle, the projection engine, and the container platform—all based on the simple primitive of immutable, content-addressed resources.

The Container Store

Universal Resource Storage

The container store is the foundation of the Hologram platform. It is a universal storage system where all resources exist as immutable, content-addressed data without inherent type or structure.

This document establishes the principles, capabilities, and properties of the container store from first principles.

First Principles

Resources as Undifferentiated Data

A **resource** is any sequence of bytes stored in the container store. Resources have no inherent type, schema, or semantic meaning. A JSON document, a binary executable, an image file, a text log—all exist as resources without distinction.

The store does not interpret, validate, or classify resources at write time. It stores bytes and provides mechanisms for retrieval and reference. Meaning emerges later through projection, not at storage time.

This principle enables universal applicability—any data can be stored, and any interpretation can be applied through projections.

Content-Addressed Identity

Every resource is identified by a **Content Identifier (CID)**, a cryptographic hash computed from the resource's complete content.

The CID function has these properties:

- **Deterministic**: The same content always produces the same CID
- **Unique**: Different content produces different CIDs (with cryptographic collision resistance)
- **Opaque**: The CID reveals nothing about content structure or meaning
- **Immutable**: Content cannot change without changing the CID

CIDs provide **intrinsic identity**—identity derived from content itself rather than assigned externally. This makes resources self-verifying: retrieving a resource by CID guarantees you receive exactly the content that produced that CID.

Immutability

Resources are **write-once**. Once stored, a resource's content never changes. The CID permanently identifies that specific content.

If content needs to change, a new resource with new content is written, producing a new CID. The original resource remains unchanged and available.

Immutability provides:

- **Reliable references**: A CID always refers to the same content
- **Historical record**: All versions persist as distinct resources
- **Concurrent access**: No coordination needed for reads (content never changes)
- **Distribution**: Resources can move between stores with identity intact

References as CIDs

Resources can reference other resources by including CIDs in their content. A JSON resource might contain:

```
{
  "spec": "cid:abc123...",
  "interface": "cid:def456...",
  "documentation": "cid:789ghi..."
}
```

These references form a **directed graph** where nodes are resources and edges are CID references. The graph is immutable—references never change—but new resources can reference existing resources, extending the graph.

The reference graph enables:

- **Aggregation**: Collecting related resources by following references
- **Composition**: Building complex structures from simple resources
- **Deduplication**: Multiple resources can reference the same resource
- **Versioning**: New versions reference previous versions

Store Capabilities

The container store provides four fundamental capabilities.

1. Store

Operation: Given content (bytes), compute the CID and store the content indexed by CID.

Properties:

- Idempotent: Storing identical content multiple times produces the same CID and stores content once
- Atomic: Content is either fully stored or not stored (no partial writes)
- Durable: Once stored, content persists until explicitly deleted

Result: The CID identifying the stored resource.

This operation makes resources available for retrieval and projection.

2. Retrieve

Operation: Given a CID, retrieve the corresponding resource content.

Properties:

- Deterministic: The same CID always retrieves the same content
- Verified: Retrieved content can be hashed to verify it matches the CID

- Complete: Resources are retrieved entirely (no partial retrieval in base operation)

Result: The complete content of the resource, or an indication that the CID is not present in the store.

This operation makes stored resources accessible.

3. Reference

Operation: Parse a resource's content to extract CIDs that reference other resources.

Properties:

- Format-dependent: Different content types encode references differently (JSON uses strings, binary formats use specific fields)
- Transitive: Following references from referenced resources builds the reference graph
- Graph-forming: References create a directed graph structure over resources

Result: A set of CIDs referenced by the resource.

This operation enables graph traversal and aggregation.

4. Query

Operation: Identify resources matching specified criteria (content patterns, reference relationships, metadata constraints).

Properties:

- Set-valued: Queries return zero or more matching resources
- Composable: Query results can be inputs to other queries
- Projection-enabling: Queries identify resources to aggregate into containers

Result: A set of CIDs for resources matching the query criteria.

This operation enables projections to identify relevant resources.

Store Properties

These capabilities provide foundational properties that the projection engine and platform rely upon.

Deduplication

Identical content produces the same CID. If the same content is stored multiple times, it occupies storage space once but can be referenced by multiple CID references.

This eliminates redundancy—shared resources (common dependencies, standard interfaces, repeated data) exist once in the store regardless of how many containers reference them.

Integrity

Retrieved content can be verified by computing its hash and comparing to the CID. Any corruption, tampering, or transmission error is detectable.

This makes the store **trustless**—you can retrieve resources from untrusted sources and verify integrity cryptographically.

Immutability

Resources never change after being written. This provides:

- **Stable references**: CIDs are reliable permanent identifiers
- **Historical completeness**: All versions of evolving data persist as separate resources
- **Conflict-free replication**: Immutable resources can be copied between stores without coordination

Content Equivalence

If two stores contain resources with the same CID, the content is identical. This enables:

- **Store synchronization**: Comparing CID sets identifies missing resources
- **Distributed storage**: Resources can live in different stores while maintaining identity
- **Caching**: Resources can be cached anywhere without invalidation concerns

Addressability

Every resource has a globally unique identifier (its CID) that remains valid across stores, networks, and time. Resources can be:

- Moved between storage backends without changing identity
- Cached locally while referencing remote stores
- Shared through CIDs without transferring content
- Archived and restored while maintaining references

Graph Structure

The reference capability creates an immutable directed graph:

- **Nodes** are resources identified by CID
- **Edges** are references (CID values in resource content)
- **Properties**:
  - Acyclic or cyclic (nothing prevents cycles)
  - Immutable (edges never change, but new nodes can be added)
  - Traversable (following edges from any node explores connected resources)

The graph structure enables:

Aggregation: Starting from a root resource, follow references to collect all related resources (a component and its documentation, tests, dependencies).

Reachability: Determine what resources are transitively referenced from a starting point, enabling garbage collection of unreferenced resources.

Versioning: New versions of resources can reference previous versions, creating version chains in the graph.

Composition: Complex containers are projections over subgraphs—collecting specific resources based on reference patterns.

Implementation Independence

The container store is defined by its capabilities and properties, not by a specific implementation.

Valid store implementations include:

Filesystem: Resources as files in content-addressed directories, references parsed from file contents.

Relational Database: Resources as BLOBs in tables, CIDs as primary keys, references extracted and indexed for query performance.

Object Storage: Resources as objects in cloud storage (S3, Azure Blob), CIDs as object keys.

Distributed Hash Table: Resources distributed across nodes in a DHT, CID-based retrieval and replication.

Hybrid: Filesystem for local working set, database for indexing and queries, object storage for archival.

The platform operates identically regardless of backend. This allows:

- Choosing storage appropriate to scale and access patterns
- Migrating between backends without changing platform semantics
- Using multiple backends simultaneously (local cache + remote archive)

Store Metadata

The store may maintain metadata about resources beyond their content:

- **Storage timestamp**: When the resource was written to this store
- **Access patterns**: How frequently the resource is retrieved
- **Reference count**: How many other resources reference this CID
- **Size**: Byte count of resource content

This metadata is **not part of the resource** and does not affect the CID. It assists with store management (garbage collection, caching, performance optimization) but is not visible to projections.

Metadata is store-specific and not preserved when resources move between stores.

Garbage Collection

Since resources are immutable and referenced by CID, unused resources can accumulate. **Garbage collection** identifies and removes resources that are no longer reachable from any root.

Roots are resources designated as starting points (active component definitions, running instances, materialized views). Resources reachable by following references from roots are retained. Unreachable resources can be deleted.

Garbage collection is a store management operation, not a platform operation. It operates on the store's reference graph without affecting the projection engine or platform semantics.

The Store as Foundation

The container store provides the invariant foundation for the platform:

- All resources exist in the store
- All resources are content-addressed and immutable
- All resources are referenceable by CID
- All resources are queryable for projection

From this foundation, we build the projection-emission cycle: projections identify and aggregate resources from the store, execution produces results, and emissions write new resources back to the store.

The next document, **The Projection-Emission Cycle**, describes how the platform operates on the container store through continuous cycles of projection and emission.

The Projection-Emission Cycle

The Fundamental Operating Principle

The Hologram platform operates through a continuous cycle of **projection** and **emission**:

Project → Execute → Emit → Store → Project…

This cycle is the fundamental mechanism by which the platform creates containers, executes operations, and evolves over time. Every interaction with Hologram—creating components, running instances, materializing views, executing operations—follows this cycle.

This document establishes how the projection-emission cycle works and why it provides a universal operating model for the platform.

The Four Phases

Phase 1: Project

Projection identifies and aggregates resources from the container store to form a container.

A projection definition specifies:

- **Query criteria**: Which resources to identify (by content patterns, reference relationships, metadata)
- **Aggregation rules**: How to collect related resources (following references, grouping by properties)
- **Transformation**: How to structure aggregated resources into a container

The projection engine evaluates the definition against the store, producing a **container**—a structured collection of resources with defined relationships and purpose.

Projection is **read-only**. It examines the store but does not modify it. The same projection evaluated multiple times against an unchanged store produces the same container.

Phase 2: Execute

Execution operates on the projected container to produce results.

Execution might:

- Validate that resources satisfy constraints
- Transform resources into different formats
- Run computation using resources as inputs
- Coordinate between multiple containers
- Generate new content from container resources

Execution consumes projected resources and produces results. Results might be:

- Status information (success, failure, validation results)
- Transformed content (compiled artifacts, rendered documentation)
- Generated data (logs, metrics, computed results)

- New resource definitions (component updates, instance state)

Execution is **isolated**. It operates on projected resources without directly accessing the store. This isolation enables:

- Reproducibility: Same inputs produce same results
- Testability: Execution can be tested with mock containers
- Distribution: Execution can occur anywhere with the projected resources

Phase 3: Emit

Emission writes execution results back to the container store as new resources.

Each result is:

1. Serialized to bytes
2. Hashed to produce a CID
3. Stored in the container store indexed by CID

Emitted resources become immediately available for future projections. They exist in the store alongside resources from previous operations, forming an accumulated knowledge base.

Emission is **additive**. New resources are added; existing resources are never modified. This preserves complete history—every emission creates a new snapshot while previous snapshots remain accessible.

Phase 4: Store

Storage persists emitted resources durably in the container store.

The store:

- Indexes resources by CID for retrieval
- Updates queryable metadata for projection
- Extends the reference graph with new resources
- Maintains integrity guarantees (immutability, deduplication)

Once stored, resources are available for projection. The cycle completes and can begin again—new projections can include newly emitted resources.

The Complete Cycle

Chaining these phases creates a continuous cycle:

1. A client requests an operation (create component, run instance, materialize view)
2. **Project**: The engine projects resources needed for the operation
3. **Execute**: The operation runs using projected resources
4. **Emit**: Results are emitted as new resources
5. **Store**: Emitted resources are persisted
6. The client can request further operations, which project the newly stored resources

Each cycle adds to the store. Over time, the store accumulates:

- Component definitions

- Instance states
- Operation logs
- Materialized views
- Application data
- User content

All emitted as immutable, content-addressed resources available for projection.

Why This Cycle Works

Universal Applicability

The same cycle applies to every platform operation:

Creating a component definition:

- Project: Identify artifact resources (spec, interface, docs, tests)
- Execute: Validate artifacts against schemas, compute component structure
- Emit: Write component definition resource
- Store: Persist component definition

Running an instance:

- Project: Identify component definition and dependencies
- Execute: Run the component's defined computation
- Emit: Write logs, state snapshots, application outputs
- Store: Persist instance emissions continuously

Materializing a view:

- Project: Identify resources matching view criteria
- Execute: Transform resources into view format
- Emit: Write materialized view resource
- Store: Persist view for efficient access

Every operation is a projection-emission cycle with operation-specific execution logic.

Composability

Cycles can chain. One operation's emissions become another operation's projection inputs:

- Create component definition → emit component resource
- Create instance from component → project component, emit instance resource
- Instance runs → emit logs and state
- Materialize log view → project logs, emit view
- Query log view → project view, return results

Each operation builds on previous emissions. Complex workflows emerge from composing simple projection-emission cycles.

Self-Evolution

The platform itself is defined through projections. Improving the platform means:

- Defining new projection definitions (new container types, new operations)

- Emitting those definitions as resources
- The engine projects the new definitions and incorporates them

The platform evolves through the same projection-emission cycle used for applications. There is no separate “platform update” mechanism—platform and application use the same primitives.

Auditability

Every emission is immutable and content-addressed. The store contains complete history:

- What resources were projected (recorded in operation logs)
- What execution occurred (recorded in execution logs)
- What results were emitted (the emitted resources themselves)
- When operations occurred (store metadata)

This provides full audit trails without additional logging infrastructure. The projection-emission cycle inherently records its own history.

Reproducibility

Given the same projected resources and execution logic, the same results are emitted. This enables:

- **Deterministic operations**: Re-running projections produces consistent results
- **Testing**: Mock stores with known resources enable testing projection-emission cycles
- **Debugging**: Reproducing issues by projecting the same historical resources

Immutability of resources guarantees that historical projections remain reproducible—the resources never change.

Examples Across the Platform

Component CRUD Operations

Create:

- Project: Artifact resources submitted by client
- Execute: Validate artifacts, build component structure
- Emit: Component definition resource
- Store: Component available for projection

Read:

- Project: Component definition and referenced resources
- Execute: Assemble complete component container
- Emit: None (read-only operation, but query results could be cached as emissions)
- Store: No new resources (unless caching results)

Update:

- Project: Existing component definition, new artifact resources
- Execute: Validate changes, build updated component
- Emit: New component definition resource (original unchanged)

- Store: Updated component available, original remains for versioning

Delete:

- Project: Component definition, dependency references
- Execute: Verify no dependencies, mark component as deleted
- Emit: Deletion marker or updated catalog
- Store: Component no longer appears in catalogs (but resources remain for history)

Instance Lifecycle

Start:

- Project: Component definition
- Execute: Initialize instance state, begin execution
- Emit: Instance resource with initial state
- Store: Instance available for monitoring

Run:

- Project: Instance state, component definition
- Execute: Continue instance execution
- Emit: Logs, metrics, application data, state updates
- Store: Continuous stream of emitted resources

Stop:

- Project: Running instance resource
- Execute: Graceful shutdown
- Emit: Final state, shutdown logs
- Store: Instance state preserved

Restart:

- Project: Stopped instance resource
- Execute: Reinitialize from last state
- Emit: New instance resource (new start)
- Store: New instance lifecycle begins

View Materialization

Materialize:

- Project: Resources matching view criteria
- Execute: Transform to view format, aggregate
- Emit: Materialized view resource
- Store: View available for efficient queries

Refresh:

- Project: Current view, new resources since last materialization
- Execute: Update view with new data
- Emit: Updated view resource
- Store: Fresh view available

Query:

- Project: Materialized view resource
- Execute: Filter/search within view
- Emit: Query results (potentially cached)
- Store: Results available for reuse

The Generative Property

Each emission adds to the store. Over time, the store grows richer:

- More components → more capabilities
- More instances → more application data
- More logs → better observability
- More views → more ways to understand the system

This is **generative**—the system becomes more capable through use. Every operation contributes to the knowledge base available for future projections.

The projection-emission cycle creates a **flywheel effect**: more emissions enable richer projections, which enable more sophisticated operations, which produce richer emissions.

Invariants

The cycle maintains critical invariants:

Store Immutability: Resources never change after emission. History is preserved.

Projection Purity: Projections don't modify the store. They're reproducible queries.

Emission Atomicity: Resources are either fully emitted or not emitted (no partial writes).

CID Integrity: Emitted resources are correctly identified by content hash.

Reference Validity: CID references in emitted resources refer to actual stored resources.

These invariants enable reasoning about the system—projections are safe queries, emissions are durable writes, the store is a reliable foundation.

Contrast with Traditional Systems

Traditional systems often separate:

- **Data storage** (databases, filesystems)
- **Computation** (application logic)
- **Output** (write operations, side effects)

Hologram unifies these through the projection-emission cycle:

- Storage, computation, and output are phases of a single cycle
- All operations follow the same pattern
- The boundary between platform and application blurs (both use projection-emission)

This unification provides consistency—learning one operation teaches the pattern for all operations.

Next Steps

The projection-emission cycle operates on containers. The next document, **Container Projections**, describes how projections transform undifferentiated resources into containers with structure, meaning, and purpose.

Container Projections

From Resources to Containers

Resources in the container store exist as undifferentiated, content-addressed data without inherent structure or meaning. **Projections** transform resources into **containers**—structured collections with defined purpose, relationships, and semantics.

This document establishes how projections work, what containers are, and how the projection mechanism provides universal containment.

What is a Container?

A **container** is a structured view of resources produced by a projection. Containers have:

Identity: A container is identified by the projection that produced it and the query parameters used (which resources, what time, what filters).

Content: A container contains or references specific resources from the store.

Structure: A container organizes resources according to the projection definition (aggregation rules, transformation logic).

Purpose: A container serves a defined role (component definition, running instance, documentation view, test suite).

Behavior: A container may define operations that can be performed with its resources (validate, execute, transform).

Containers are **ephemeral projections** or **materialized resources**:

- **Ephemeral**: The container exists during projection evaluation, used immediately, then discarded
- **Materialized**: The container is emitted as a resource to the store for persistent access

Projection Mechanism

A projection transforms resources into containers through three stages:

Stage 1: Identification

Query evaluation identifies resources in the store matching specified criteria.

Query criteria can include:

- **Content patterns**: Resources containing specific data (JSON with particular fields, text matching patterns)
- **CID references**: Resources referenced by known CIDs
- **Reference relationships**: Resources that reference or are referenced by other resources
- **Metadata constraints**: Resources with specific metadata (stored during a time range, accessed frequently)

The result is a **resource set**—the CIDs of matching resources.

Stage 2: Aggregation

Aggregation collects related resources by following reference relationships.

Starting from the resource set, aggregation:

- Follows CID references to retrieve related resources
- Applies traversal rules (depth limits, reference type filters)
- Collects transitive closure (all reachable resources) or specific subgraphs
- Groups resources by relationships or properties

The result is an **aggregated resource graph**—resources and their relationships.

Stage 3: Transformation

Transformation structures aggregated resources into container format.

Transformation might:

- Extract specific fields from resources
- Combine multiple resources into unified structure
- Apply formatting or encoding conversions
- Compute derived values from resource content
- Order or filter based on container requirements

The result is a **container**—the final structured collection ready for use.

Projection Definitions

A **projection definition** is itself a resource in the store that specifies how to project resources into containers.

Projection definitions contain:

- **Query specification**: How to identify resources
- **Aggregation rules**: How to collect related resources
- **Transformation logic**: How to structure the container
- **Conformance requirements**: What constraints aggregated resources must satisfy

Because projection definitions are resources, they can be:

- Versioned (new projection definitions with different CIDs)
- Composed (projection definitions that reference other projection definitions)
- Projected (meta-projections that analyze or transform projection definitions)

This creates **recursive projections**—the system projects its own projection definitions.

Container Types Through Projection

Different projection definitions create different container types. The platform provides base projection definitions for standard container types:

Component Container:

- Identifies: Resources with component namespace
- Aggregates: Spec, interface, documentation, tests, dependencies

- Transforms: Structured component definition
- Purpose: Reusable capability definition

Instance Container:

- Identifies: Resources with instance identifier
- Aggregates: Component definition, instance state, runtime context
- Transforms: Executable instance
- Purpose: Running component with state

Interface Container:

- Identifies: Resources with interface definitions
- Aggregates: Method signatures, type definitions, contracts
- Transforms: Interface specification
- Purpose: Contract between components

Documentation Container:

- Identifies: Documentation resources
- Aggregates: Docs by namespace, cross-references
- Transforms: Human-readable documentation
- Purpose: Understanding and guidance

Test Container:

- Identifies: Test resources
- Aggregates: Test cases, expected results, validation logic
- Transforms: Executable test suite
- Purpose: Validation and verification

View Container:

- Identifies: Resources matching view criteria
- Aggregates: According to view definition
- Transforms: Materialized view format
- Purpose: Efficient access to projected data

Each container type is defined by its projection definition. New container types are created by emitting new projection definitions.

Projection Composition

Projections can compose—one projection can use another's results as input.

Nested Projection: A component container projects interface containers for each interface it references. The interface projections run as sub-projections of the component projection.

Sequential Projection: Create component → project component → create instance → project instance. Each projection's output becomes input to the next operation.

Parallel Projection: Multiple projections execute concurrently over the same store. A documentation view and a test view project the same components simultaneously.

Composition enables building complex containers from simple projections without implementing complex monolithic projection logic.

Projection Parameters

Projections can accept parameters that customize their behavior:

Resource Filters: Project only resources matching specific criteria (namespace, time range, content patterns).

Depth Limits: Control how deeply to follow references during aggregation.

Transformation Options: Enable optional transformations (include/exclude certain fields, format preferences).

Temporal Constraints: Project resources as they existed at a specific point in time (time-travel queries).

Parameterized projections enable reusing projection definitions for different contexts without creating multiple definition variants.

The Projection Graph

Executing multiple projections creates a **projection graph**:

- **Nodes**: Containers produced by projections
- **Edges**: Dependencies where one projection uses another's results

The projection graph shows relationships between containers:

- Component containers reference interface containers
- Instance containers reference component containers
- View containers reference multiple other container types

The projection graph is distinct from the resource reference graph:

- Resource graph: Immutable CID references between resources
- Projection graph: Ephemeral relationships between projected containers

Containers vs Resources

Key distinctions:

Resources:

- Immutable content in the store
- Identified by CID (content hash)
- No inherent structure or meaning
- Permanent (until garbage collected)

Containers:

- Structured projections of resources
- Identified by projection definition + parameters
- Defined structure and purpose
- Ephemeral (exist during use) or materialized (emitted as resources)

A resource can participate in multiple containers. The same documentation resource might appear in:

- A component container (as component docs)
- A documentation view container (as part of searchable docs)
- A tutorial container (as example content)

Projection determines how the resource appears and what role it serves.

Projection Purity

Projections are **pure queries**—they do not modify the store or have side effects.

Properties:

- **Deterministic**: Same store state and parameters produce same container
- **Repeatable**: Projecting multiple times yields consistent results
- **Concurrent**: Multiple projections can run simultaneously without interference
- **Cacheable**: Projection results can be cached and reused

Purity enables:

- **Testing**: Projections can be tested with known resource sets
- **Debugging**: Reproducing projections for investigation
- **Optimization**: Caching projection results for performance

Side effects (creating resources, executing stateful operations) occur during the **Execute** phase of the projection-emission cycle, not during projection itself.

Temporal Projections

Because resources are immutable, the store preserves history. **Temporal projections** project resources as they existed at a previous time.

A temporal projection:

1. Identifies resources that existed at the specified time (based on store metadata)
2. Aggregates using only references that existed at that time
3. Produces a container representing historical state

This enables:

- **Versioning**: View previous versions of components or instances
- **Audit**: Understand what resources were available when operations occurred
- **Debugging**: Reproduce historical state to investigate issues

Temporal projections work because resources never change—a CID always refers to the same content, regardless of when you project it.

Dynamic Projections

Some projections depend on runtime state or external context:

Instance Projections: Project current instance state, which changes as the instance emits new resources.

Live View Projections: Project continuously, incorporating new resources as they're emitted.

Contextual Projections: Include user-specific or environment-specific resources (user permissions, deployment configuration).

Dynamic projections are still pure queries against the store's current state, but that state evolves through emissions, causing projection results to change over time.

Projection Performance

Projection performance depends on:

Store Query Efficiency: How quickly the store can identify matching resources (indexing, caching).

Graph Traversal: How many references must be followed during aggregation (depth, branching factor).

Transformation Complexity: How much computation is required to structure the container.

Resource Size: How much data must be retrieved and processed.

Optimization strategies:

- **Materialized Views**: Pre-compute and emit common projections as resources
- **Incremental Updates**: Update views with only new/changed resources rather than full reprojection
- **Lazy Aggregation**: Follow references only as needed rather than complete traversal
- **Query Optimization**: Index store metadata for fast resource identification

Projection as Lens

Projections act as **lenses** that reveal different aspects of the store:

- A component lens reveals reusable definitions
- An instance lens reveals running state
- A documentation lens reveals human-readable content
- A dependency lens reveals relationship graphs

The same underlying resources appear differently through different lenses. There is no "true" view—all containers are valid projections serving different purposes.

This multiplicity enables flexible organization: structure resources however projections require without committing to a single schema or hierarchy.

Next Steps

Projections are defined and executed by the **Projection Engine**, which interprets projection definitions and orchestrates the identification-aggregation-transformation pipeline.

The next section, Part II: The Engine, details how the projection engine works, the projection language for defining projections, and how emissions create new resources for future projections.

The Projection Engine

Executing Projections

The **projection engine** is the component that interprets projection definitions and executes them against the container store to produce containers.

This document describes how the engine works, its architecture, execution model, and how it orchestrates the projection-emission cycle.

Engine Responsibilities

The projection engine:

Interprets Projection Definitions: Reads projection definition resources from the store and parses them into executable instructions.

Executes Queries: Evaluates query criteria against the store to identify matching resources.

Traverses References: Follows CID references to aggregate related resources according to aggregation rules.

Applies Transformations: Structures aggregated resources into containers according to transformation logic.

Manages Execution Context: Maintains state during projection execution (visited resources, depth tracking, intermediate results).

Coordinates Composition: Orchestrates nested and sequential projections, managing dependencies between them.

Handles Errors: Validates constraints, reports failures, ensures projections complete successfully or fail cleanly.

The engine is the **interpreter** for the projection language—it gives operational meaning to declarative projection definitions.

Execution Model

Projection Request

A projection execution begins with a **request** specifying:

- **Projection Definition CID**: Which projection to execute
- **Parameters**: Values for parameterized queries or transformations
- **Context**: Additional information (user identity, environment, temporal constraints)

The engine retrieves the projection definition resource, validates it, and begins execution.

Three-Phase Execution

The engine executes projections in three phases, corresponding to the projection mechanism:

Phase 1: Query Evaluation (Identification)

The engine evaluates query criteria from the projection definition against the container store.

Steps:

1. Parse query specification from projection definition
2. Apply parameters to customize query
3. Execute query against store (using store's query capability)
4. Receive resource set (CIDs of matching resources)
5. Validate resource set meets minimum requirements (if specified)

Result: A set of CIDs identifying resources to aggregate.

Phase 2: Reference Traversal (Aggregation)

The engine follows references from the resource set to collect related resources.

Steps:

1. Parse aggregation rules from projection definition
2. Initialize traversal state (visited set, depth counter)
3. For each resource in the set:
    - Retrieve resource content from store
    - Extract CID references from content
    - Determine which references to follow (based on rules)
    - Recursively retrieve referenced resources
    - Track relationships in aggregated graph
4. Apply depth limits and cycle detection
5. Validate aggregated resources meet conformance requirements

Result: An aggregated resource graph with content and relationships.

Phase 3: Transformation (Structuring)

The engine transforms the aggregated graph into container format.

Steps:

1. Parse transformation logic from projection definition
2. Extract required fields from resources
3. Combine resources according to structure rules
4. Apply formatting and encoding conversions
5. Compute derived values
6. Order and filter based on container requirements
7. Validate final container structure

Result: A structured container ready for use.

Execution Context

The engine maintains execution context throughout projection:

Visited Resources: Set of CIDs already retrieved during traversal, preventing redundant

retrieval and cycle detection.

Depth Tracking: Current traversal depth for enforcing depth limits.

Error State: Collection of validation failures or constraint violations encountered during execution.

Intermediate Results: Partial containers or computed values used across phases.

Parameter Bindings: Resolved parameter values used throughout execution.

Context is isolated per projection—concurrent projections do not share context or interfere with each other.

Query Evaluation

The engine delegates query evaluation to the container store's query capability, but orchestrates the process:

Query Translation: Convert projection definition's query specification into store-specific query format.

Query Execution: Invoke store query, handling pagination or streaming for large result sets.

Result Filtering: Apply additional filters that the store cannot evaluate (complex content patterns, computed predicates).

Result Validation: Ensure query results meet cardinality constraints (minimum/maximum resource counts).

The engine abstracts store-specific query details—the same projection definition works with different store implementations that have different query capabilities.

Reference Traversal Algorithm

Reference traversal follows a **controlled graph traversal**:

```
function traverse(rootCIDs, rules, maxDepth):
    visited = empty set
    toVisit = queue of (CID, depth=0) from rootCIDs
    graph = empty aggregated graph

    while toVisit not empty:
        (cid, depth) = toVisit.dequeue()

        if cid in visited:
            continue
        if depth > maxDepth:
            continue

        visited.add(cid)
        content = store.retrieve(cid)
        graph.addResource(cid, content)

        references = extractReferences(content)
        for refCID in references:
            if shouldFollow(refCID, rules):
                toVisit.enqueue((refCID, depth+1))
                graph.addEdge(cid, refCID)

    return graph
```

Key aspects:

Visited Tracking: Prevents retrieving the same resource multiple times and handles reference cycles.

Depth Limits: Controls traversal extent, preventing unbounded graph exploration.

Selective Following: Rules determine which references to follow (by reference type, target resource properties).

Breadth-First: Queue-based traversal explores resources level by level (though depth-first is also valid).

Transformation Pipeline

Transformation applies a sequence of operations to the aggregated graph:

Extraction: Select specific resources or fields from the graph.

Combination: Merge multiple resources into unified structures.

Computation: Derive values from resource content (counts, aggregations, computed fields).

Formatting: Convert encodings or representations (JSON to human-readable, binary to text).

Filtering: Remove resources or fields that don’t meet criteria.

Ordering: Sort resources by specified properties.

Each operation takes the current intermediate result and produces a new intermediate result, building toward the final container structure.

Conformance Validation

During aggregation, the engine validates that resources conform to requirements specified in the projection definition.

Schema Validation: Resources match expected structure (JSON schema, type definitions).

Cardinality Constraints: Required resources are present, optional resources handled correctly.

Reference Integrity: Referenced CIDs exist in the store and are reachable.

Semantic Constraints: Application-specific rules (version compatibility, naming conventions).

Validation failures:

- **Hard Failures**: Abort projection, return error (missing required resources)
- **Soft Failures**: Record warning, continue (optional resource not found)
- **Accumulation**: Collect all failures, report at completion

The engine provides detailed error information identifying which resources or constraints failed.

Projection Composition

The engine handles composed projections where one projection uses another's results.

Nested Projection: During Phase 2 (aggregation), encounter a reference that requires sub-projection:

1. Suspend current projection
2. Initiate sub-projection with referenced projection definition
3. Execute sub-projection to completion
4. Incorporate sub-projection result into current aggregated graph
5. Resume current projection

Sequential Projection: One projection completes, results emitted to store, next projection projects the emitted resources:

1. Execute first projection
2. Emit results to store
3. Store returns CIDs of emitted resources
4. Execute second projection with emitted CIDs as query parameters
5. Continue chain

Parallel Projection: Multiple projections execute concurrently:

1. Engine spawns multiple execution contexts
2. Each context executes its projection independently
3. Contexts do not share state (pure queries, no interference)
4. Results collected as projections complete

Composition enables building complex containers from simple, reusable projection definitions.

Caching and Optimization

The engine can optimize repeated projections:

Result Caching: Store completed container results, keyed by (projection definition CID, parameters). Repeated identical projections return cached results.

Partial Caching: Cache intermediate results (query results, aggregated graphs) for reuse in similar projections.

Incremental Evaluation: For dynamic projections over evolving stores, update previous results with only new/changed resources rather than complete re-evaluation.

Query Planning: Analyze projection definitions to optimize execution order (evaluate selective queries first, parallelize independent operations).

Caching is transparent—cached projections return identical results to fresh evaluation, just faster.

Error Handling

The engine provides structured error information:

Query Failures: Store query errors (malformed query, store unavailable).

Retrieval Failures: Missing CIDs, corrupted resources, access denied.

Validation Failures: Resources don't conform to schema or constraints.

Transformation Failures: Cannot convert format, computation errors, invalid structure.

Resource Errors: Out of memory, timeout, execution limits exceeded.

Each error includes:

- Error type and description
- Affected resource CIDs
- Projection definition context (which phase, which rule)
- Suggestions for resolution (if applicable)

Errors propagate: sub-projection failures cause parent projection failure, providing complete failure trace.

Execution Guarantees

The engine provides guarantees:

Purity: Projections do not modify the store or have side effects. Same input always produces same output.

Isolation: Concurrent projections do not interfere. Each has independent execution context.

Completeness: Successful projection returns complete container meeting all conformance requirements. Partial results are not returned.

Determinism: Given identical store state and parameters, projection produces identical results.

Atomicity: Projection either completes successfully or fails cleanly. No partial state persists.

These guarantees enable reliable reasoning about projections and composing them confidently.

Engine Architecture

The engine consists of modular components:

Definition Parser: Reads projection definition resources, validates syntax, produces executable representation.

Query Executor: Translates and executes queries against store, handles results.

Traversal Manager: Implements reference traversal algorithm, manages visited tracking and depth limits.

Transformation Engine: Applies transformation pipeline, validates intermediate results.

Conformance Validator: Checks resources against schemas and constraints.

Context Manager: Maintains execution context, handles error state, provides telemetry.

Composition Coordinator: Manages nested and sequential projections, tracks dependencies.

Components interact through well-defined interfaces, enabling implementations in different languages or with different optimization strategies.

Engine Implementation Independence

Like the container store, the projection engine is defined by its behavior, not implementation.

Valid engine implementations might:

- Interpret projection definitions directly (interpreter)
- Compile projection definitions to native code (compiler)
- Distribute execution across multiple nodes (distributed engine)
- Specialize for specific projection types (optimizing engine)

The platform works identically with any conforming engine implementation.

Meta-Projections

The engine can project its own projection definitions—examining, analyzing, or transforming them.

Projection Analysis: Project all projection definitions, analyze query patterns, identify optimization opportunities.

Projection Validation: Project projection definitions, validate they’re well-formed and reference valid resources.

Projection Composition: Project multiple projection definitions, synthesize new composed projection definition.

This reflexivity enables the platform to reason about and improve its own projection capabilities.

Next Steps

The projection engine interprets projection definitions written in the **projection language**. The next document describes this language—how conformance requirements specify projection instructions, and how projections are defined as resources in the store.

The Projection Language

Conformance as Projection Instructions

The **projection language** is how projections are defined as resources in the container store. What appears as “conformance requirements” in component definitions is actually a **declarative language** for specifying how to project resources into containers.

This document describes the projection language—its syntax, semantics, and how conformance requirements serve as projection instructions.

The Conformance Model

Traditional systems use conformance for validation: “does this data match this schema?”

Hologram inverts this: **conformance defines how to project**. A conformance requirement specifies:

- What resources to identify
- How to aggregate related resources
- What structure the resulting container must have
- What transformations to apply

Conformance requirements are **projection instructions** interpreted by the projection engine to produce containers.

Projection Definition Structure

A projection definition is a resource containing:

Identity: The projection type and namespace (e.g., “hologram.component”, “hologram.interface”).

Query Specification: Criteria for identifying resources to project.

Conformance Requirements: Rules for aggregating and structuring resources into containers.

Schema Definitions: Expected structure for aggregated resources (JSON schemas, type definitions).

Transformation Rules: How to convert aggregated resources into final container format.

Metadata: Versioning, documentation, dependencies on other projection definitions.

The definition itself is stored in the container store, identified by CID. Different versions of a projection definition have different CIDs.

Query Specification

The query specification identifies which resources to project.

Namespace Queries

Resources are often organized by **namespace**—a hierarchical naming scheme embedded in resource content.

A namespace query identifies resources belonging to a specific namespace:

- “hologram.component” → all component definition resources
- “hologram.interface” → all interface resources
- “application.user” → application-specific user resources

The engine queries the store for resources with matching namespace fields.

Content Pattern Queries

Queries can match resource content patterns:

- Resources containing specific fields
- Resources with field values matching predicates
- Resources matching text patterns
- Resources of specific content types (JSON, binary, text)

Content pattern queries enable flexible resource identification beyond namespace conventions.

Reference Queries

Queries can identify resources based on reference relationships:

- Resources referenced by a known resource
- Resources that reference a known resource
- Resources transitively reachable from a starting point
- Resources forming specific graph patterns

Reference queries enable graph-based resource selection.

Temporal Queries

Queries can include temporal constraints:

- Resources stored within a time range
- Resources as they existed at a specific point in time
- Resources modified after a certain timestamp

Temporal queries enable historical projections and time-travel.

Composite Queries

Queries can combine multiple criteria:

- Namespace AND content pattern
- Reference relationship OR namespace
- Temporal constraint AND content pattern

Boolean composition enables precise resource identification.

Conformance Requirements

Conformance requirements define how to aggregate and structure identified resources.

Required Resources

Specify resources that **must** be present for valid projection:

A component conformance requirement might specify:

- “spec” resource (component specification)
- “interface” resource (interface definition)

Missing required resources cause projection failure.

Optional Resources

Specify resources that **may** be present:

A component might have:

- Optional “documentation” resource
- Optional “examples” resource

Missing optional resources do not cause failure but affect container structure.

Cardinality Constraints

Specify how many of each resource type:

- Exactly one: “spec” (1)
- One or more: “tests” (1..n)
- Zero or more: “dependencies” (0..n)
- Zero or one: “documentation” (0..1)

Cardinality violations cause projection failure.

Reference Requirements

Specify how resources must reference each other:

An interface conformance requirement might specify:

- Must reference a “component” (parent relationship)
- Must be referenced by component’s “interface” field (consistency)

Reference requirements ensure graph structure integrity.

Schema Requirements

Specify structure for each resource type:

Each required or optional resource has an associated schema (often JSON Schema) defining:

- Required and optional fields
- Field types and formats
- Valid value ranges

- Structural constraints

Resources that don’t match schemas cause validation failure.

Semantic Requirements

Specify application-specific constraints:

- Naming conventions (namespace follows pattern)
- Version compatibility (dependencies at compatible versions)
- Uniqueness (no duplicate names within namespace)
- Business rules (specific field relationships)

Semantic requirements are evaluated during aggregation or transformation.

Transformation Rules

Transformation rules specify how aggregated resources become the final container structure.

Field Extraction

Extract specific fields from resources:

- From “spec” resource, extract “namespace”, “version”, “description”
- From “interface” resource, extract “methods”

Extracted fields populate container properties.

Resource Combination

Combine multiple resources into unified structure:

- Merge all “test” resources into “tests” array
- Combine “documentation” resources by section

Combination creates hierarchical container structure.

Computed Fields

Derive values from resource content:

- Count of dependencies
- Hash of concatenated test resources
- Latest timestamp across all resources

Computed fields add metadata to containers.

Format Conversion

Convert resource encodings:

- Binary to base64 text
- JSON to human-readable formatted text

- Markdown to HTML

Format conversion adapts resources to container requirements.

Ordering and Filtering

Order resources by property (alphabetically, by timestamp, by dependency order).

Filter resources by criteria (exclude deprecated, include only active).

Ordering and filtering refine container contents.

Parameterized Projections

Projection definitions can include parameters that customize behavior at execution time.

Parameter Declaration

Define parameters the projection accepts:

- “namespace” (string): Which namespace to project
- “include_optional” (boolean): Whether to include optional resources
- “max_depth” (integer): Maximum reference traversal depth

Parameters are declared in the projection definition with types and constraints.

Parameter Binding

At execution time, the engine binds parameter values:

- From client request (explicit parameter values)
- From context (environment variables, user identity)
- From defaults (declared in projection definition)

Bound parameters are substituted into query specification and transformation rules.

Parameterized Queries

Parameters customize queries:

Instead of hard-coding “hologram.component”, parameterize as “{namespace}”.

At execution, “{namespace}” is replaced with the bound value.

This enables reusing projection definitions across different namespaces.

Conditional Rules

Rules can be conditional on parameters:

IF “include_optional” THEN aggregate optional documentation resources.

Conditional rules enable flexible projection behavior.

Projection Composition in the Language

The projection language supports defining composed projections.

Sub-Projection References

A projection definition can reference other projection definitions:

A component projection might specify:

- Project “hologram.interface” for interface resources
- Project “hologram.documentation” for documentation resources

The engine executes sub-projections and incorporates results into the parent projection.

Nested Conformance

Conformance requirements can nest:

A component requires:

- “interface” conforming to “hologram.interface” projection
- “documentation” conforming to “hologram.documentation” projection

Nested conformance creates projection hierarchies.

Projection Inheritance

Projection definitions can extend other projection definitions:

“hologram.advanced_component” extends “hologram.component”:

- Inherits all base requirements
- Adds additional requirements
- Overrides specific transformation rules

Inheritance enables specialization without duplication.

Language Semantics

Declarative Nature

The projection language is **declarative**—it describes **what** to project, not **how** to project.

The engine determines execution strategy (query optimization, traversal algorithm, caching).

This enables engine evolution without changing projection definitions.

Purity

Projection definitions are **pure specifications**—they don’t include imperative code with side effects.

No “execute this function”, only “aggregate these resources with these rules”.

Purity enables analysis, optimization, and reasoning about projections.

Composability

Projections compose cleanly:

- Reference other projections without tight coupling
- Inherit and extend without duplicating logic
- Parameterize without creating explosion of variants

Composability enables building complex projections from simple components.

Versioning

Projection definitions are versioned through CID:

- New version → new resource → new CID
- Old versions remain available (immutability)
- Resources can reference specific projection versions

Versioning enables evolution without breaking existing projections.

Schema Language

The projection language includes or references a schema language for defining resource structure.

JSON Schema Integration

The platform uses JSON Schema as the primary schema language:

- Mature, widely adopted standard
- Rich constraint language
- Tool support for validation

JSON schemas are resources in the store, referenced by projection definitions.

Schema Composition

Schemas can reference other schemas:

- Common type definitions shared across schemas
- Schema inheritance through “allOf”, “oneOf”
- Modular schema construction

Schema composition mirrors projection composition.

Schema Versioning

Like projection definitions, schemas are versioned via CID.

Projection definitions reference specific schema versions, ensuring stable validation over time.

Language Extensions

The projection language is extensible—new query types, conformance requirements, and transformation rules can be added.

Custom Query Types

Define new query types as resources:

- Graph pattern matching queries
- Probabilistic similarity queries
- Machine learning-based classification queries

The engine loads custom query implementations and applies them.

Custom Validators

Define new validation rules as resources:

- Application-specific business rules
- Cross-resource consistency checks
- External system integration (check against external API)

Custom validators plug into the conformance validation phase.

Custom Transformations

Define new transformations as resources:

- Domain-specific format conversions
- Complex computations (rendering, compilation)
- Aggregations specific to application needs

Custom transformations extend the transformation pipeline.

Extensions are themselves resources, making the language **self-extensible**.

Conformance as Lens Definition

Viewing conformance as projection instructions reveals its true role:

Traditional View: “This resource conforms to this schema” (validation).

Hologram View: “Project resources using these conformance requirements to create this container type” (projection).

Conformance requirements are **lens definitions**—they define how to view the store through a particular lens to see a particular type of container.

Different conformance requirements (lenses) applied to the same resources produce different containers (views).

Language Bootstrap

The projection language is defined using itself:

- “hologram.projection” is a projection definition for projecting projection definitions

- It specifies how to identify, aggregate, and structure projection definition resources
- The engine uses “hologram.projection” to understand projection definitions

This self-description enables the language to evolve—new language features are expressed as updates to the “hologram.projection” definition.

Next Steps

Projections produce containers through the **Project → Execute** phases. The **Emit** phase produces new resources from execution results.

The next document, **The Emission Model**, describes how containers emit resources during execution, what types of emissions occur, and how emissions integrate back into the store for future projections.

The Emission Model

How Containers Emit Resources

Emission is the process by which execution results are written back to the container store as new resources. Emissions complete the projection-emission cycle, making execution results available for future projections.

This document describes the emission model—what emissions are, how they work, emission types, and how they integrate into the platform.

What is an Emission?

An **emission** is the creation of a new resource in the container store as a result of executing a projection.

Emissions have these properties:

Content: The data being emitted (bytes, structured data, references to other resources).

Identity: The CID computed from the content, uniquely identifying the emission.

Immutability: Once emitted, the resource never changes (consistent with store immutability).

Addressability: The emission is immediately available for retrieval and projection via its CID.

Atomicity: The emission is either fully stored or not stored (no partial emissions).

Emissions transform ephemeral execution results into durable resources.

The Emission Process

Phase 1: Result Generation

During the **Execute** phase of the projection-emission cycle, operations produce results:

- Validated component definitions
- Instance state snapshots
- Log entries
- Computed views
- Application data
- Metadata

Results exist initially as ephemeral execution artifacts (in-memory structures, temporary files).

Phase 2: Serialization

Results are **serialized** into a canonical byte representation:

Structured Data: Converted to canonical JSON (sorted keys, consistent formatting).

Binary Data: Used as-is (already bytes).

References: Embedded as CID strings within serialized content.

Metadata: Included in serialized representation or as separate associated resource.

Serialization produces deterministic byte sequences—the same result always serializes identically.

Phase 3: Content Addressing

The serialized content is **hashed** to compute the CID:

- Apply cryptographic hash function (SHA-256 or similar)
- Produce fixed-length hash digest
- Format as CID (typically “cid:” prefix + hex digest)

Content addressing ensures identical content produces identical CIDs (deduplication) and enables integrity verification.

Phase 4: Storage

The serialized content and CID are **stored** in the container store:

- Store maps CID to content
- Content becomes retrievable via the CID
- Store updates indexes and metadata
- Reference graph is extended if content contains CID references

Once stored, the emission is durable and available for projection.

Phase 5: Reference Return

The emission process returns the CID to the caller (the operation that produced the result).

The caller can:

- Return the CID to the client (for user-initiated operations)
- Use the CID in further processing (as input to subsequent projections)
- Emit additional resources that reference the new CID

Returned CIDs enable chaining—one emission becomes input to the next operation.

Emission Types

Different operations emit different types of resources.

Definition Emissions

Component Definitions: Results of creating or updating components.

Content:

- Component specification
- References to interface, documentation, tests, dependencies

- Version information
- Namespace and metadata

These emissions define reusable capabilities available for instantiation.

State Emissions

Instance State: Snapshots of running instance state.

Content:

- Current variable values
- Execution position
- Pending tasks
- Resource allocations
- State timestamp

State emissions enable persistence, recovery, and debugging.

Log Emissions

Log Entries: Records of events during execution.

Content:

- Timestamp
- Log level (info, warning, error)
- Message
- Context (operation, resource, user)
- References to related resources

Log emissions provide observability and audit trails.

Data Emissions

Application Data: User content, computed results, generated artifacts.

Content:

- Application-specific structured data
- Files or binary artifacts
- User input or user-generated content
- Computed aggregations or transformations

Data emissions are the primary output of running instances.

View Emissions

Materialized Views: Pre-computed projections stored for efficient access.

Content:

- Query results
- Aggregated resources
- Transformed representations

- View metadata (freshness timestamp, query parameters)

View emissions optimize repeated queries.

Metadata Emissions

Operational Metadata: Information about operations themselves.

Content:

- Operation start/end timestamps
- Resources projected during operation
- Execution duration
- Success/failure status
- Error details

Metadata emissions enable monitoring and analysis.

Emission Streams

Some operations produce **continuous emissions** rather than single emissions.

Running Instances: Emit logs, state snapshots, and application data continuously while running.

Live Views: Emit updated view resources as underlying resources change.

Event Streams: Emit event resources as events occur (user actions, external triggers).

Emission streams are sequences of individual emissions, each producing a distinct resource with distinct CID.

Buffering and Batching

High-frequency emission streams may use buffering:

- Collect multiple emission candidates in memory
- Batch serialize and store together
- Reduce store overhead from individual tiny emissions

Batching is transparent—logically each emission is distinct, even if physically stored in a batch.

Stream Termination

Emission streams terminate when:

- Instance stops (graceful or forced)
- View subscription is cancelled
- Event source is closed

Termination may produce a final emission marking stream completion.

Emission References

Emitted resources can reference other resources via CID references.

Parent References: A log emission references the instance that produced it.

Dependency References: A component definition references interface and documentation resources.

Sequential References: A new component version references the previous version.

Aggregation References: A view emission references all resources it aggregated.

References form the resource graph, enabling:

- Tracing relationships (find all logs for an instance)
- Versioning (follow version chains)
- Garbage collection (determine reachability)

Emission Deduplication

If emitted content is identical to an existing resource, the existing CID is returned without storing duplicate content.

This occurs naturally from content addressing:

1. Serialize result
2. Compute CID
3. Check if CID exists in store
4. If yes, return existing CID (no storage needed)
5. If no, store content and return new CID

Deduplication is automatic and transparent—callers always receive correct CID regardless of whether content was newly stored or already existed.

Emission Validation

Before emitting, the platform may validate that the result conforms to expected structure:

Schema Validation: Ensure emitted resource matches schema for its type.

Reference Integrity: Ensure referenced CIDs exist in the store.

Constraint Satisfaction: Ensure business rules or semantic constraints are met.

Validation failures prevent emission—invalid resources are not stored. The operation reports error and can retry or abort.

Emission Atomicity

Emissions are atomic operations:

All-or-Nothing: Content is either fully stored and indexed, or not stored at all (no partial writes).

Consistent State: Store remains consistent whether emission succeeds or fails.

Isolation: Concurrent emissions don’t interfere—each produces independent resource.

Atomicity enables reliable reasoning: if emission succeeds, the resource is available and correctly formed.

Emission Transactions

Operations that emit multiple related resources can use **emission transactions**:

1. Prepare all resources to emit
2. Compute all CIDs
3. Validate all resources
4. Store all resources atomically
5. Update store indexes

If any resource fails validation, the entire transaction aborts—no resources are emitted.

Transactions ensure related resources appear together (component definition with all conformance resources) or not at all.

Emission Idempotence

Repeating an emission operation with identical input produces identical output:

- Same result content
- Same CID
- Store state unchanged (deduplication)

Idempotence enables safe retries after failures—re-emitting is harmless.

Emission Observability

The platform provides visibility into emissions:

Emission Logs: Record of what was emitted, when, by what operation.

Emission Metrics: Count of emissions, size distribution, emission rate.

Emission Traces: Which projections led to which emissions (operation chains).

Observability helps understand platform behavior and debug issues.

Emissions as First-Class Resources

Emitted resources are indistinguishable from any other resources in the store:

- Same content addressing
- Same immutability
- Same projectionability
- Same lifecycle

There is no special “emission” type—emissions simply add resources to the store.

This uniformity means:

- Emissions can be projected like any resources
- Emissions can reference and be referenced by any resources
- Platform and application emissions are treated identically

The Feedback Loop

Emissions create a feedback loop:

1. **Project** resources from store
2. **Execute** operation on projected resources
3. **Emit** results as new resources to store
4. New resources become available for projection
5. Future projections include newly emitted resources
6. **Repeat**

Each cycle enriches the store. Over time:

- More components → more operations possible
- More logs → better observability
- More state → more sophisticated recovery
- More views → more efficient queries

The feedback loop makes the platform **generative**—it becomes more capable through use.

Emission Policies

The platform may define policies governing emissions:

Retention: How long emissions persist before garbage collection.

Rate Limits: Maximum emission rate to prevent store overload.

Size Limits: Maximum size of individual emissions.

Access Control: Who can emit what types of resources.

Schema Requirements: What schemas emissions must conform to.

Policies ensure store health and enforce organizational requirements.

Emission Patterns

Common emission patterns across operations:

Create-Emit: Create new resource, emit immediately (component creation).

Accumulate-Emit: Collect results over time, emit periodically (batched logs).

Transform-Emit: Project resources, transform, emit result (view materialization).

Trigger-Emit: External event triggers emission (user action creates data).

Replicate-Emit: Emit copy of resource from another store (synchronization).

Patterns provide templates for implementing new operations.

Emission and Versioning

Every emission creates a new version:

- New component definition with changes → new CID, new version

- Updated instance state → new CID, new state snapshot
- Refreshed view → new CID, new view version

Old versions remain in store (immutability). Version history is automatically preserved through distinct CIDs.

Clients can reference specific versions (by CID) or request “latest” (projection query for most recent by timestamp).

Emission Performance

Emission performance depends on:

Serialization Cost: Time to convert results to bytes.

Hashing Cost: Time to compute CID.

Store Write Latency: Time for store to persist and index resource.

Validation Cost: Time to validate before emission.

Optimization strategies:

- Efficient serialization formats
- Fast hash algorithms
- Asynchronous store writes (return CID before write completes)
- Batch emissions

High emission rate is critical for interactive applications and streaming data.

Contrast with Traditional Systems

Traditional systems separate computation output from storage:

- Computation produces results
- Separate storage operation writes results to database/filesystem
- Multiple result formats (database rows, files, caches)

Hologram unifies:

- Emission is the canonical output mechanism
- Emitted resources are immediately storable and projectable
- Single format (content-addressed resources)

Unification simplifies the platform—everything flows through the same mechanism.

Next Steps

With the projection engine executing projections and the emission model creating new resources, the platform provides a complete cycle for operations.

The next section, Part III: The Platform, describes the **Container Types** provided by the platform—the specific projection definitions and emission patterns for components, instances, interfaces, documentation, tests, views, and more.

Container Types

Platform Container Types and Emissions

The Hologram platform provides **base container types**—projection definitions and associated operations for common containment patterns. Container types define how resources are projected into containers and what emissions those containers produce.

This document describes the platform's base container types, their projection patterns, and emission characteristics.

Container Type Fundamentals

A **container type** is defined by:

Projection Definition: The conformance requirements and transformation rules for projecting resources into this container type.

Required Resources: What resources must be present for valid containers of this type.

Optional Resources: What resources may be included but aren't mandatory.

Emission Pattern: What resources containers of this type emit during their lifecycle.

Operations: What operations can be performed with containers of this type.

Container types are defined as resources in the store (projection definitions), making them extensible—new container types are added by emitting new projection definitions.

Component Containers

Purpose: Define reusable capabilities as specifications.

Projection Pattern

Query: Identify resources with component namespace (e.g., "hologram.component", "application.service").

Required Resources:

- **Spec**: Component specification (namespace, version, description, metadata)

Optional Resources:

- **Interface**: Interface definition resources
- **Documentation**: Human-readable documentation resources
- **Tests**: Test suite resources
- **Dependencies**: References to other components required
- **Build**: Build process definition
- **Manager**: Lifecycle management definition

Aggregation: Follow references from spec to collect all related resources.

Transformation: Structure into hierarchical component definition with spec as root, related resources as branches.

Emission Pattern

Creation: Emit component definition resource containing spec and references to all conformance resources.

Update: Emit new component definition resource with updated content (new CID, immutable versioning).

Deletion Marker: Emit resource marking component as deprecated or deleted (original definition remains for history).

Characteristics

- **Immutable**: Once created, component definitions don't change (updates create new versions)
- **Versioned**: Each update produces new CID, creating version chain
- **Reusable**: Can be referenced by multiple instances or other components
- **Self-Describing**: Contains all information needed to understand and use the component

Instance Containers

Purpose: Execute component definitions with runtime state.

Projection Pattern

Query: Identify resources with instance identifier (references specific component definition and instance ID).

Required Resources:

- **Component Reference**: CID of component definition to instantiate
- **Instance Identity**: Unique identifier for this instance

Optional Resources:

- **Initial State**: Starting state for the instance
- **Configuration**: Instance-specific configuration overriding defaults
- **Context**: Runtime environment information

Aggregation: Project component definition, then collect instance-specific resources.

Transformation: Combine component definition with instance state/config into executable instance container.

Emission Pattern

Initialization: Emit initial instance state resource.

Runtime: Continuous emission stream while running:

- **State Snapshots**: Periodic or event-triggered state captures
- **Log Entries**: Events, errors, info messages
- **Application Data**: Output produced by instance execution
- **Metrics**: Performance and resource utilization data

Termination: Emit final state and shutdown logs when instance stops.

Characteristics

- **Stateful**: Maintains evolving state over time through emissions
- **Ephemeral**: Can be started, stopped, removed (though emissions persist)
- **Observable**: Continuous emission stream provides visibility
- **Recoverable**: State snapshots enable restart from last known state

Interface Containers

Purpose: Define contracts between components.

Projection Pattern

Query: Identify interface definition resources (often by namespace or referenced by component).

Required Resources:

- **Interface Specification**: Methods, parameters, return types, error conditions

Optional Resources:

- **Schema Definitions**: Type definitions for parameters/returns
- **Documentation**: Description of interface semantics
- **Examples**: Usage examples

Aggregation: Collect interface spec and related type/schema resources.

Transformation: Structure into interface definition with methods and types clearly defined.

Emission Pattern

Definition: Emit interface definition resource.

Versioning: Emit new interface versions (semantic versioning for compatibility tracking).

Conformance Validation: Implementations emit validation resources demonstrating conformance to interface.

Characteristics

- **Contract-Defining**: Specifies expectations between components
- **Versioned**: Interface evolution tracked through versions
- **Reusable**: Multiple components can implement same interface
- **Validation-Enabling**: Enables checking component compatibility

Documentation Containers

Purpose: Provide human-readable explanations and guides.

Projection Pattern

Query: Identify documentation resources (by namespace, by reference from components).

Required Resources:

- **Documentation Content**: Markdown, HTML, or other human-readable format

Optional Resources:

- **Examples**: Code examples, usage demonstrations
- **Diagrams**: Visual representations
- **Cross-References**: Links to related documentation

Aggregation: Collect documentation resources and related examples/diagrams.

Transformation: Structure into navigable documentation with sections, cross-references resolved.

Emission Pattern

Creation: Emit documentation resource.

Updates: Emit updated documentation (versioned with component versions).

Rendered Views: Emit transformed formats (HTML from Markdown, PDF from LaTeX).

Characteristics

- **Human-Oriented**: Designed for human consumption, not machine execution
- **Cross-Linked**: References other documentation, components, interfaces
- **Multi-Format**: Can be projected into different representations
- **Versioned**: Evolves with component versions

Test Containers

Purpose: Validate component behavior and correctness.

Projection Pattern

Query: Identify test resources (by namespace, by component reference).

Required Resources:

- **Test Cases**: Individual test definitions (inputs, expected outputs, assertions)

Optional Resources:

- **Test Fixtures**: Shared setup/teardown logic
- **Test Data**: Input datasets for tests
- **Validation Rules**: Expected behavior specifications

Aggregation: Collect test cases, fixtures, and data into test suite.

Transformation: Structure into executable test suite.

Emission Pattern

Suite Execution: Emit test results:

- **Pass/Fail Status**: Per-test and overall results
- **Execution Logs**: Details of test execution
- **Coverage Data**: What code/behavior was tested
- **Performance Metrics**: Test execution time

Continuous Testing: Emit updated test results as tests re-run.

Characteristics

- **Executable**: Test containers run to produce results
- **Validation-Focused**: Purpose is to verify correctness
- **Evolving**: New tests added as components evolve
- **Result-Producing**: Emissions are test outcomes

Manager Containers

Purpose: Control lifecycle of instances and components.

Projection Pattern

Query: Identify manager definition resources (lifecycle management logic).

Required Resources:

- **Lifecycle Operations**: Definitions for start, stop, restart, update operations

Optional Resources:

- **Health Checks**: Logic for determining instance health
- **Scaling Rules**: Criteria for starting/stopping multiple instances
- **Recovery Policies**: How to handle failures

Aggregation: Collect manager operations and policies.

Transformation: Structure into manager definition with operation handlers.

Emission Pattern

Operation Execution: Emit operation results:

- **State Transitions**: Instance moved from stopped to running
- **Operation Logs**: What actions were taken
- **Status Updates**: Current manager state

Health Monitoring: Emit health check results periodically.

Characteristics

- **Control-Oriented**: Manages other containers' lifecycles
- **Policy-Driven**: Behavior defined by policies in projection
- **Reactive**: Responds to state changes and external requests
- **Status-Emitting**: Continuously updates status

View Containers

Purpose: Materialized projections for efficient access.

Projection Pattern

Query: Identify resources matching view criteria (flexible, view-specific).

Required Resources: Varies by view type (any resources the view aggregates).

Optional Resources: Varies by view type.

Aggregation: Follows view-specific aggregation rules (might be complex multi-step).

Transformation: Transform to view format (often denormalized, indexed).

Emission Pattern

Materialization: Emit view resource (pre-computed projection result).

Refresh: Emit updated view resource as underlying resources change.

Index Emissions: Emit index structures for efficient view queries.

Characteristics

- **Denormalized**: Often aggregate and flatten for query performance
- **Refreshable**: Can be updated incrementally or fully rebuilt
- **Query-Optimized**: Structured for fast access patterns
- **Derived**: Always derivable from underlying resources (can be rebuilt)

Dependency Containers

Purpose: Express relationships between components.

Projection Pattern

Query: Identify dependency resources (references between components).

Required Resources:

- **Dependent Component**: Component requiring dependencies
- **Required Component**: Component being depended upon
- **Dependency Type**: Nature of dependency (interface, data, service)

Optional Resources:

- **Version Constraints**: Compatible version ranges
- **Optional Flags**: Whether dependency is required or optional

Aggregation: Build dependency graph by following dependency references.

Transformation: Structure into graph or list of dependencies.

Emission Pattern

Dependency Declaration: Emit dependency resource linking components.

Resolution Results: Emit resolved dependency sets (satisfying version constraints).

Conflict Warnings: Emit warnings if dependency conflicts detected.

Characteristics

- **Graph-Forming**: Dependencies create directed graph over components
- **Constraint-Carrying**: Express version and compatibility requirements
- **Resolvable**: Dependency resolution algorithms operate on these containers
- **Validation-Enabling**: Enable checking for circular dependencies, conflicts

Build Containers

Purpose: Define transformation from source to executable artifacts.

Projection Pattern

Query: Identify build definition resources.

Required Resources:

- **Build Steps**: Sequence of transformation operations
- **Source References**: Input resources to build process

Optional Resources:

- **Build Environment**: Required tools, dependencies
- **Build Configuration**: Parameters, flags, options

Aggregation: Collect build definition and referenced sources.

Transformation: Structure into executable build plan.

Emission Pattern

Build Execution: Emit build results:

- **Built Artifacts**: Output resources from build
- **Build Logs**: Details of build process
- **Build Status**: Success/failure
- **Build Metadata**: Build timestamp, environment, inputs

Characteristics

- **Transformation-Focused**: Converts inputs to outputs
- **Repeatable**: Same inputs and build definition produce same outputs
- **Artifact-Producing**: Primary emission is built artifacts
- **Traceable**: Build logs provide full traceability

Log Containers

Purpose: Structured logging and event records.

Projection Pattern

Query: Identify log entry resources (by source, by time range, by level).

Required Resources:

- **Log Entries**: Individual log records

Optional Resources:

- **Context Information**: Associated resource references, user identity

Aggregation: Collect logs by query criteria (temporal, source-based).

Transformation: Structure into queryable log collection (often chronological).

Emission Pattern

Log Creation: Emit individual log entry resources continuously.

Log Aggregation: Emit aggregated log views (hourly, daily summaries).

Alert Emissions: Emit alert resources when logs match alert criteria.

Characteristics

- **Time-Ordered**: Logs have timestamps, often queried chronologically
- **High-Volume**: Can produce many emissions rapidly
- **Queryable**: Often projected into views for searching
- **Retention-Sensitive**: May be garbage collected after retention period

Extending Container Types

New container types are created by emitting new projection definitions.

Process:

1. Define conformance requirements (what resources, what structure)
2. Define transformation rules (how to structure container)
3. Define emission patterns (what the container emits)
4. Emit projection definition resource to store
5. Platform recognizes new container type, enables projecting it

This extensibility means the platform is **open**—applications can define domain-specific container types without modifying platform code.

Container Type Composition

Container types compose naturally:

- Components reference interfaces and documentation
- Instances reference components
- Views aggregate any container type
- Managers control instances

Composition creates rich ecosystems where container types work together.

Next Steps

View containers deserve detailed treatment due to their role as optimized, persistent projections. The next document, **Platform Views**, explores views in depth—how they're materialized, refreshed, and used for efficient access to projected data.

Platform Views

Materialized Container Projections

Views are persistent materializations of projections—pre-computed container projections emitted as resources for efficient repeated access. Views optimize common queries by storing projection results rather than re-projecting on every access.

This document describes platform views, their materialization process, refresh strategies, and role in the platform architecture.

What is a View?

A **view** is a container produced by projecting resources, then emitted as a resource itself for reuse.

Views have dual nature:

As Containers: Views are projected containers—results of executing projection definitions against the store.

As Resources: Views are emitted to the store as resources, making them durable and projectable themselves.

This duality enables:

- Expensive projections computed once, accessed many times
- Projections of projections (meta-views)
- Version control of view states (immutable CIDs)
- Distribution of views between stores

View Materialization

Materialization is the process of executing a projection and emitting the result as a view resource.

Materialization Process

Phase 1: Projection Execution

Execute the view's projection definition:

1. Query store to identify resources
2. Aggregate related resources by following references
3. Transform aggregated resources into view structure
4. Validate view satisfies conformance requirements

Result: A view container (ephemeral, in-memory).

Phase 2: Emission

Emit the view container as a resource:

1. Serialize view container to canonical format
2. Compute CID from serialized content

3. Store view resource in container store
4. Index view for efficient retrieval

Result: A view resource with CID, available for projection.

Phase 3: Catalog Update

Update the view catalog:

1. Record view existence (CID, view type, parameters)
2. Record materialization timestamp
3. Record source resources (what was projected)
4. Index by view type and parameters for discovery

Result: View discoverable through catalog queries.

View Types

Different view types serve different purposes.

Catalog Views

Purpose: Index available resources for discovery.

Content:

- List of all components with metadata (namespace, version, description)
- List of all instances with status (running, stopped)
- List of available interfaces
- Component dependency graph

Refresh: Periodic or on-demand when components added/removed.

Usage: Clients query catalog views to discover what's available.

Component Views

Purpose: Denormalized component information for fast access.

Content:

- Complete component definition
- Inlined interface definitions (rather than just references)
- Rendered documentation (HTML from Markdown)
- Test result summaries

Refresh: When component is updated.

Usage: Displaying component details without traversing references.

Instance Views

Purpose: Runtime state and observability.

Content:

- Current instance state
- Recent logs (last N entries)
- Performance metrics
- Resource utilization

Refresh: Continuous or frequent (real-time view).

Usage: Monitoring dashboards, debugging, operations.

Log Views

Purpose: Aggregated logs for searching and analysis.

Content:

- Log entries grouped by instance/component/time
- Indexed by timestamp, level, source
- Aggregated statistics (error counts, rates)

Refresh: Continuous as logs emitted, periodic aggregation.

Usage: Log search, troubleshooting, audit.

Dependency Views

Purpose: Visualize and analyze component relationships.

Content:

- Complete dependency graph
- Transitive dependency closure
- Reverse dependencies (what depends on this)
- Dependency conflict detection

Refresh: When components or dependencies change.

Usage: Understanding system structure, impact analysis.

Documentation Views

Purpose: Searchable, navigable documentation.

Content:

- All documentation resources indexed by topic/component
- Cross-references resolved
- Search index for full-text queries
- Table of contents and navigation structure

Refresh: When documentation updated.

Usage: Documentation portals, search interfaces.

Spec Views

Purpose: Export component definitions for version control.

Content:

- Component definitions in filesystem-friendly format
- Directory structure mirroring component namespaces
- Human-readable JSON formatting

Refresh: On-demand when user wants to sync to filesystem.

Usage: Git commits, external tool integration, review.

View Refresh Strategies

Views can become stale as underlying resources change. **Refresh strategies** determine when and how to update views.

On-Demand Refresh

View refreshed when explicitly requested by client.

Advantages:

- No background computation cost
- View freshness controlled by client

Disadvantages:

- First access after staleness incurs refresh latency
- Clients must know when refresh is needed

Applicable: Views where staleness is acceptable (historical snapshots, archived data).

Periodic Refresh

View refreshed at regular intervals (hourly, daily).

Advantages:

- Bounded staleness (at most one period old)
- Simple scheduling
- Predictable refresh cost

Disadvantages:

- May refresh unnecessarily (if no changes)
- Still can be stale between refreshes

Applicable: Catalog views, dashboards, reports.

Incremental Refresh

View updated with only new/changed resources since last refresh.

Advantages:

- Lower cost than full refresh (only process changes)
- Keeps view fresher

- Scales better with large views

Disadvantages:

- More complex implementation
- Requires tracking changes

Applicable: Log views, event streams, time-series data.

Invalidation-Based Refresh

View refreshed when underlying resources change.

Advantages:

- View always fresh (no staleness)
- No unnecessary refreshes (only when needed)

Disadvantages:

- Requires change tracking/notification
- May refresh too frequently if resources change often

Applicable: Critical views (security, availability), small views with fast refresh.

Lazy Refresh

View refreshed on first access after invalidation.

Advantages:

- Combines invalidation detection with on-demand refresh
- No refresh cost if view not accessed

Disadvantages:

- First post-invalidation access has latency

Applicable: Frequently invalidated but infrequently accessed views.

View Versioning

Because views are resources with CIDs, they're naturally versioned.

Each refresh produces a new view resource with a new CID. The old view remains (immutability).

This enables:

Temporal Queries: Project the view as it existed at a specific time (by CID or timestamp).

Consistency: Multiple queries against the same view CID see identical data (no concurrent modifications).

Auditing: Complete history of view states preserved.

Rollback: If refresh produces incorrect view, use previous version.

The view catalog tracks view versions, maintaining:

- Current view CID (latest refresh)
- Previous view CIDs (history)
- Refresh timestamps
- Underlying resource versions

View Consistency

Views can have different consistency guarantees.

Eventually Consistent Views

View may lag behind store state. Projecting a view might return data from before recent emissions.

Advantages: Lower refresh cost, better performance.

Disadvantages: Stale data possible.

Applicable: Most views (catalogs, dashboards, documentation).

Strongly Consistent Views

View reflects all emissions up to query time.

Advantages: No staleness, accurate data.

Disadvantages: Higher cost (frequent refresh or projection without materialization).

Applicable: Critical operational views (instance status, security).

Snapshot Consistent Views

View reflects store state at specific point in time (consistent snapshot).

Advantages: Consistent data (all resources from same timepoint).

Disadvantages: Intentionally stale (snapshot timestamp < current time).

Applicable: Reports, analytics, historical analysis.

View Queries

Views are resources, so they can be projected. **View queries** project views to extract information.

Direct View Projection

Retrieve entire view resource by CID, use contents directly.

Fast (single resource retrieval), but returns entire view.

Filtered View Projection

Project view, apply filters to select subset.

More flexible than direct retrieval, enables searching within view.

View Composition

Project multiple views, combine results.

Enables answering queries that span view types (components with instances + logs).

Parameterized View Projection

Views can be parameterized (by namespace, time range, filters).

Projection binds parameters, retrieves specific view variant.

Enables view families (component view for each namespace) without storing every variant.

View Caching

Views serve as caches—materialized projections avoid re-computation.

Cache Hit: Projecting a view retrieves materialized result (fast).

Cache Miss: View doesn't exist or is stale, requires refresh (slow).

Cache Invalidation: Determine when view is stale and needs refresh.

Standard caching challenges apply:

- How to detect staleness
- When to refresh
- How to balance freshness vs cost

View refresh strategies are cache invalidation policies.

Views and Storage Backend

Views can be stored in the same backend as resources or in specialized storage:

Same Backend: Views are resources like any others (simple, consistent).

Specialized Storage: Views in database optimized for queries (relational DB, search index).

Hybrid: Frequently accessed views in fast storage, others in standard storage.

The view model is independent of storage—views are logically resources regardless of physical storage.

View Materialization Cost

Materializing views has costs:

Projection Cost: Time and resources to execute projection (query, aggregate, transform).

Storage Cost: Space to store materialized view resources.

Refresh Cost: Overhead of detecting changes and updating views.

Consistency Cost: Ensuring view matches store state.

Optimization strategies:

Selective Materialization: Only materialize frequently accessed views.

Incremental Refresh: Update rather than rebuild views.

Approximate Views: Accept approximate results for lower cost (sampling, estimation).

Lazy Materialization: Defer materialization until first access.

Views as Projections of Projections

Views are resources, so views can project other views—**meta-views**.

A summary view might project detail views:

- Daily log view projects hourly log views
- System overview view projects component and instance views
- Dashboard view projects multiple operational views

Meta-views enable hierarchical aggregation and composition without accessing underlying resources repeatedly.

View Definition

View definitions are projection definitions with additional metadata:

Projection Definition: How to project resources into view.

Refresh Strategy: When and how to refresh view.

Retention Policy: How long to keep old view versions.

Indexing Hints: What fields to index for efficient queries.

Access Patterns: Expected query patterns (optimize for these).

View definitions are resources in the store, enabling new view types to be added by emitting new view definitions.

Views and Garbage Collection

Views are derived—they can always be rebuilt from underlying resources.

This makes views candidates for garbage collection:

- Remove old view versions beyond retention policy
- Remove unused views to free storage
- Rebuild views from resources if needed

Views provide the **denormalization** tradeoff: storage cost vs query performance.

The Spec View

The **spec view** deserves special attention—it's the view of component definitions formatted for filesystem storage and version control.

Purpose: Enable git-based workflow for component definitions.

Content: Component definition resources serialized as filesystem hierarchy:

- Directory per component namespace
- Files per resource type (spec, interface, docs, tests)
- Human-readable formatting (pretty-printed JSON)
- Index files for navigation

Refresh: On-demand when developer wants to commit changes.

Bidirectional: Resources can be read from spec view (import from git) or written to spec view (export for git).

The spec view bridges Hologram (content-addressed store) with traditional version control systems (filesystem + git).

Next Steps

Views are consumed through operations—projection executions that read resources, potentially materialize views, and emit results.

The next document, **Platform Operations**, describes the operations provided by the platform—how they use projection-emission cycles, what inputs they require, and what outputs they produce.

Platform Operations

Operations as Projections That Emit

Operations are the platform's mechanisms for interacting with resources through projection-emission cycles. Every operation follows the pattern: project resources, execute logic, emit results.

This document describes platform operations, their structure, and how they embody the projection-emission cycle.

Operation Fundamentals

An **operation** is a named, executable procedure that:

Projects: Identifies and aggregates resources from the store based on operation inputs.

Executes: Performs computation, validation, transformation, or coordination using projected resources.

Emits: Writes results back to the store as new resources.

Returns: Provides operation outcome to the caller (success/failure, emitted resource CIDs).

Operations are the **verbs** of the platform—the actions that transform store state.

Operation Structure

Every operation has:

Identity: Operation name and type (e.g., "component.create", "instance.start").

Inputs: Parameters provided by caller (resource CIDs, values, options).

Projection Phase: How the operation identifies and aggregates resources from the store.

Execution Phase: What computation the operation performs on projected resources.

Emission Phase: What resources the operation writes back to the store.

Outputs: What the operation returns to caller (status, emitted CIDs, error information).

This structure maps directly to the projection-emission cycle.

Component Operations

Operations for managing component definitions.

Component Create

Purpose: Define a new component from artifacts.

Inputs:

- Artifact resources (spec, interface, docs, tests, etc.)

- Component namespace
- Version information

Projection Phase:

- Project artifact resources by CID
- Project component model (hologram.component definition) to understand conformance requirements

Execution Phase:

- Validate artifacts against schemas
- Check conformance requirements satisfied
- Build component definition structure
- Assign unique component identifier

Emission Phase:

- Emit component definition resource
- Emit component index entry (for catalog)

Outputs:

- Component definition CID
- Success/failure status
- Validation messages

Component Read

Purpose: Retrieve component definition and related resources.

Inputs:

- Component namespace or CID

Projection Phase:

- Project component definition resource
- Project referenced resources (interface, docs, tests per request)

Execution Phase:

- Assemble complete component container
- Resolve references
- Apply formatting/transformation if requested

Emission Phase:

- None (read-only operation)
- May cache projection result for performance

Outputs:

- Component definition
- Referenced resources
- Metadata

Component Update

Purpose: Modify existing component definition.

Inputs:

- Component namespace or CID
- Updated artifact resources
- Change description

Projection Phase:

- Project existing component definition
- Project new artifact resources
- Project dependents (components depending on this one)

Execution Phase:

- Validate new artifacts
- Check backward compatibility if required
- Build updated component definition
- Verify dependents not broken

Emission Phase:

- Emit new component definition resource (new CID, immutable versioning)
- Emit change log entry
- Update component index

Outputs:

- New component definition CID
- Change summary
- Compatibility status

Component Delete

Purpose: Remove component from catalog.

Inputs:

- Component namespace or CID
- Force flag (delete even with dependents)

Projection Phase:

- Project component definition
- Project reverse dependencies (what depends on this)

Execution Phase:

- Check for dependents
- If dependents exist and not forced, fail
- Mark component as deleted

Emission Phase:

- Emit deletion marker resource
- Update component index (remove from catalog)
- Emit dependency warning if forced deletion

Outputs:

- Deletion status
- Affected dependents list

Component Validate

Purpose: Verify component definition correctness.

Inputs:

- Component namespace or CID

Projection Phase:

- Project component definition
- Project all referenced resources
- Project schemas for validation

Execution Phase:

- Validate spec against schema
- Validate conformance resources
- Check reference integrity
- Verify dependencies resolvable

Emission Phase:

- Emit validation report resource

Outputs:

- Validation result (pass/fail)
- Detailed validation messages
- Report CID

Instance Operations

Operations for managing running instances.

Instance Create

Purpose: Instantiate a component definition.

Inputs:

- Component definition CID or namespace
- Instance configuration
- Initial state (optional)

Projection Phase:

- Project component definition

- Project dependency components (transitive)
- Project runtime context

Execution Phase:

- Resolve dependencies
- Initialize instance state
- Allocate resources
- Prepare execution environment

Emission Phase:

- Emit instance resource (initial state)
- Emit instance catalog entry
- Emit initialization log

Outputs:

- Instance ID and CID
- Initialization status

Instance Start

Purpose: Begin instance execution.

Inputs:

- Instance ID

Projection Phase:

- Project instance resource
- Project component definition
- Project dependencies

Execution Phase:

- Load instance state
- Begin execution per component definition
- Start emission streams (logs, state, data)

Emission Phase:

- Emit instance state update (status: running)
- Emit startup logs
- Begin continuous emission stream

Outputs:

- Start status
- Instance runtime information

Instance Stop

Purpose: Halt instance execution.

Inputs:

- Instance ID
- Graceful timeout (optional)

Projection Phase:

- Project running instance resource

Execution Phase:

- Signal instance to stop
- Wait for graceful shutdown (up to timeout)
- Force stop if timeout exceeded
- Finalize state

Emission Phase:

- Emit final instance state (status: stopped)
- Emit shutdown logs
- Emit execution summary

Outputs:

- Stop status
- Final state CID

Instance Restart

Purpose: Stop and start instance.

Inputs:

- Instance ID
- Restart options

Projection Phase:

- Project instance resource

Execution Phase:

- Execute stop operation
- Execute start operation

Emission Phase:

- Emissions from stop and start operations
- Emit restart log entry

Outputs:

- Restart status
- New instance state CID

Instance Inspect

Purpose: Retrieve instance state and status.

Inputs:

- Instance ID

Projection Phase:

- Project instance resource
- Project recent logs
- Project component definition

Execution Phase:

- Assemble current instance state
- Gather runtime information
- Compute uptime and metrics

Emission Phase:

- None (read-only)
- May cache inspection result

Outputs:

- Instance state
- Status information
- Logs and metrics

View Operations

Operations for managing materialized views.

View Materialize

Purpose: Execute projection and emit result as view resource.

Inputs:

- View definition CID
- Projection parameters
- Materialization options

Projection Phase:

- Project view definition
- Execute view’s projection (project resources per view definition)

Execution Phase:

- Transform projection result to view format
- Apply indexing or optimization
- Validate view structure

Emission Phase:

- Emit view resource
- Emit view catalog entry
- Update view index

Outputs:

- View CID
- Materialization timestamp
- Statistics (resources processed, view size)

View Refresh

Purpose: Update existing view with current store state.

Inputs:

- View CID or identifier
- Refresh strategy (full, incremental)

Projection Phase:

- Project existing view
- Project new/changed resources since last refresh
- Project view definition

Execution Phase:

- Determine changes since last materialization
- Apply incremental updates or full rebuild
- Validate updated view

Emission Phase:

- Emit refreshed view resource (new CID)
- Update view catalog with new version
- Emit refresh log

Outputs:

- New view CID
- Changes applied
- Refresh statistics

View Query

Purpose: Execute query against materialized view.

Inputs:

- View CID or identifier
- Query parameters (filters, search terms)

Projection Phase:

- Project view resource
- Apply query filters

Execution Phase:

- Search or filter within view
- Compute results
- Apply pagination or limits

Emission Phase:

- None (read-only)
- May emit query result cache

Outputs:

- Query results
- Result metadata (count, timing)

Artifact Operations

Operations for staging resources before component creation.

Artifact Submit

Purpose: Stage a resource for later use in component creation.

Inputs:

- Artifact content
- Artifact type (spec, interface, docs, etc.)

Projection Phase:

- None (no existing resources needed)

Execution Phase:

- Validate artifact content
- Check artifact type valid

Emission Phase:

- Emit artifact resource to store
- Record in artifact staging area

Outputs:

- Artifact CID
- Staging status

Manifest Submit

Purpose: Create component from staged artifacts.

Inputs:

- Manifest listing artifact CIDs and types
- Component namespace

Projection Phase:

- Project staged artifact resources
- Project component model

Execution Phase:

- Execute component create operation using artifacts
- Clear artifact staging area

Emission Phase:

- Emissions from component create
- Clear staging artifacts

Outputs:

- Component definition CID
- Component creation status

Validation Operations

Operations for verifying correctness.

Schema Validate

Purpose: Validate resource against schema.

Inputs:

- Resource CID
- Schema CID or type

Projection Phase:

- Project resource to validate
- Project schema definition

Execution Phase:

- Apply schema validation
- Collect validation errors/warnings

Emission Phase:

- Emit validation report

Outputs:

- Validation result
- Error details

Reference Validate

Purpose: Verify reference integrity.

Inputs:

- Resource CID
- Traversal depth

Projection Phase:

- Project resource

- Project all referenced resources (recursively)

Execution Phase:

- Check all CID references exist
- Verify no dangling references
- Detect reference cycles if problematic

Emission Phase:

- Emit reference validation report

Outputs:

- Integrity status
- Missing references
- Cycle information

Catalog Operations

Operations for discovering resources.

Catalog List

Purpose: List available resources by type.

Inputs:

- Resource type (components, instances, views)
- Filters (namespace pattern, status)

Projection Phase:

- Project catalog view for resource type
- Apply filters

Execution Phase:

- Extract matching entries
- Sort and paginate

Emission Phase:

- None (read-only)
- May emit query result cache

Outputs:

- Resource list
- Metadata

Catalog Search

Purpose: Search resources by content or metadata.

Inputs:

- Search query (text, patterns)
- Resource types to search

Projection Phase:

- Project search indexes or catalog views
- Execute search query

Execution Phase:

- Rank results by relevance
- Apply pagination

Emission Phase:

- None (read-only)
- May emit search result cache

Outputs:

- Search results
- Relevance scores

Dependency Operations

Operations for managing component relationships.

Dependency Resolve

Purpose: Compute transitive dependencies for a component.

Inputs:

- Component CID or namespace

Projection Phase:

- Project component definition
- Project dependency view or build graph

Execution Phase:

- Traverse dependency references
- Build transitive closure
- Check for conflicts or cycles

Emission Phase:

- Emit dependency resolution result
- Emit dependency graph

Outputs:

- Resolved dependencies (all required components)
- Resolution status
- Conflict information

Operation Composition

Operations can compose:

Sequential: Component Create → Instance Create → Instance Start (build workflow).

Parallel: Multiple Instance Start operations concurrently.

Conditional: Instance Stop → if graceful failed → Instance Force Stop.

Transactional: Artifact Submit (multiple) → Manifest Submit (atomic component creation).

Composition enables complex workflows from simple operations.

Operation Atomicity

Operations are atomic with respect to emissions:

- All emissions succeed or all fail
- Store remains consistent regardless of operation outcome
- Partial failures result in complete rollback

This enables reliable operations even with concurrent access.

Operation Idempotence

Many operations are idempotent:

- Repeating with same inputs produces same result
- Component Create with identical artifacts produces same CID
- View Materialize with same parameters produces same view

Idempotence enables safe retries after failures.

Operation Observability

All operations emit logs and metadata:

- Operation start/end timestamps
- Resources projected
- Execution duration
- Success/failure status
- Emitted resource CIDs

This provides complete audit trail and debugging information.

Next Steps

Operations are invoked through the **Platform API**—the client interface for interacting with the platform. The next document describes the API model, request/response patterns, and how clients use the API to perform operations.

Platform API

Client Interaction Model

The **Platform API** is how clients interact with the Hologram platform—submitting operations, retrieving results, subscribing to emissions, and querying resources. The API provides a uniform interface across different client implementations.

This document describes the API model, request/response patterns, and client interaction principles.

API Principles

Protocol Independence

The API model is abstract—not tied to specific protocols (HTTP, gRPC, MCP, CLI).

Different client implementations provide different protocol bindings:

- **MCP Server**: Model Context Protocol for AI tool integration
- **HTTP API**: RESTful or GraphQL over HTTP
- **CLI Client**: Command-line interface
- **SDK Libraries**: Language-specific client libraries

All bindings map to the same underlying API model.

Operation-Centric

The API is organized around **operations** (component.create, instance.start, view.materialize).

Each operation is a distinct API endpoint or command with specific:

- Input parameters
- Execution semantics
- Output format
- Error conditions

Clients invoke operations and receive results.

Resource-Oriented

Operations manipulate **resources** identified by CID or namespace.

The API provides:

- Resource submission (emit new resources)
- Resource retrieval (project existing resources)
- Resource querying (identify resources matching criteria)

Resources are the nouns, operations are the verbs.

Asynchronous by Default

Many operations (instance start, view materialization) have non-trivial duration.

The API supports **asynchronous execution**:

- Submit operation request, receive operation ID
- Poll or subscribe to operation status
- Retrieve results when complete

Short operations can complete synchronously, but async is always available.

Request Model

Operation Requests

All operation requests have common structure:

Operation Identifier: Which operation to execute (e.g., “component.create”).

Parameters: Operation-specific inputs:

- Resource CIDs (references to existing resources)
- Values (strings, numbers, structured data)
- Options (flags, preferences)

Context: Metadata about the request:

- Client identity (user, service account)
- Request ID (for tracing)
- Timeout preferences
- Priority hints

Idempotency Token: Optional token ensuring repeated requests don’t duplicate operations.

Resource Submission

Submitting resources for emission:

Content: The resource data (bytes or structured data).

Metadata: Optional metadata:

- Content type (JSON, binary, text)
- Description
- Tags for categorization

Validation Options: Whether to validate before emission, what schema to use.

The API computes CID, validates (if requested), and emits to store.

Projection Requests

Requesting projection without performing full operation:

Projection Definition: Which projection to execute (by CID or type).

Parameters: Projection parameters (namespace filter, depth limit, etc.).

Format: Desired output format (structured data, human-readable, binary).

The API executes projection, returns container without emitting.

Response Model

Synchronous Responses

For operations that complete quickly:

Status: Success or failure indicator.

Result: Operation output:

- Emitted resource CIDs
- Projected resource content
- Computed values

Metadata: Execution information:

- Duration
- Resources accessed
- Warnings

Errors: If failure, detailed error information.

Asynchronous Responses

For long-running operations:

Operation ID: Unique identifier for tracking operation.

Status Endpoint: Where to check operation status.

Estimated Duration: If known, expected completion time.

Clients poll status endpoint or subscribe to status updates:

Status Updates: Periodic messages indicating progress:

- State (pending, running, completed, failed)
- Progress percentage (if computable)
- Intermediate results

Completion: Final message with full results or error.

Streaming Responses

For operations producing continuous output (instance logs, event streams):

Stream Handle: Identifier for the stream.

Stream Endpoint: Where to receive stream data.

Stream Control: Operations to pause, resume, or close stream.

Stream delivers items as they're emitted:

- Log entries
- State updates
- Events

Stream terminates on completion or explicit close.

Error Model

Error Structure

Errors provide detailed information:

Error Code: Machine-readable identifier (RESOURCE_NOT_FOUND, VALIDATION_FAILED).

Error Message: Human-readable description.

Context: Where error occurred:

- Operation phase (projection, execution, emission)
- Resource CID that caused error
- Stack trace (if applicable)

Recovery Suggestions: Possible remediation (fix schema, provide missing resource).

Related Resources: CIDs of resources involved in error.

Error Categories

Client Errors: Invalid input, malformed request, unauthorized access.

- HTTP 4xx equivalent
- Client should fix request and retry

Server Errors: Platform failure, store unavailable, internal error.

- HTTP 5xx equivalent
- Client should retry (possibly with backoff)

Resource Errors: Resource not found, validation failed, reference broken.

- Specific to Hologram model
- May require emitting new/different resources

Operation Errors: Operation-specific failures (can't start instance, dependency conflict).

- Varies by operation
- Check operation documentation for possible errors

Authentication and Authorization

Identity

Clients authenticate, establishing identity:

- User identity (human user)
- Service identity (automated client)
- Anonymous (if platform allows)

Identity is provided in request context.

Permissions

The platform enforces access control:

- Read permissions (project resources)
- Write permissions (emit resources)
- Operation permissions (execute specific operations)

Permission model is flexible:

- Resource-based (permissions per resource namespace)
- Role-based (roles grant permission sets)
- Attribute-based (contextual permissions)

Unauthorized operations return permission denied error.

Client Patterns

Common interaction patterns.

Submit-Retrieve Pattern

For two-phase operations (artifact/manifest):

1. Client submits multiple resources (artifacts)
2. API returns CIDs
3. Client submits manifest referencing CIDs
4. API executes operation (component create)
5. Client retrieves result

This pattern enables staging and validation before final operation.

Subscribe-Process Pattern

For monitoring running instances:

1. Client starts instance
2. Client subscribes to instance emission stream
3. API delivers logs, state, data as emitted
4. Client processes items as received
5. Client closes stream when done

This pattern enables real-time monitoring.

Query-Refine Pattern

For discovery:

1. Client queries catalog (broad search)

2. API returns initial results
3. Client refines query (add filters)
4. API returns refined results
5. Repeat until desired resources found

This pattern supports interactive exploration.

Batch-Process Pattern

For high-volume operations:

1. Client submits batch of operations
2. API processes concurrently
3. API returns batch results
4. Client checks each result

This pattern optimizes throughput.

API Versioning

Projection Versioning

Since operations are defined by projections in the store, API evolution is natural:

New Operations: Emit new projection definitions to store.

Operation Changes: Emit new version of projection definition (new CID).

Deprecation: Mark old projection versions as deprecated (still available).

The API surface evolves through the same projection-emission mechanism.

Protocol Versioning

Protocol bindings (HTTP, MCP) have independent versions:

- Protocol version indicates what features are available
- Older clients use older protocol versions
- Platform supports multiple protocol versions

Implementation Examples

How different client types interact:

MCP Server Client

MCP (Model Context Protocol) client for AI tools:

Request: AI tool invokes MCP tool (listComponents, validate, create).

Mapping: MCP tool maps to platform operation.

Execution: Platform executes operation.

Response: Platform returns result formatted per MCP.

Streaming: MCP server events for long operations.

HTTP API Client

HTTP client for web applications:

Request: HTTP POST to operation endpoint with JSON body.

Mapping: URL path + method indicate operation, body contains parameters.

Execution: Platform executes operation.

Response: HTTP response with JSON result.

Streaming: Server-Sent Events or WebSocket for streams.

CLI Client

Command-line client for operators:

Request: CLI command with arguments and flags.

Mapping: Command name maps to operation, arguments to parameters.

Execution: Platform executes operation.

Response: Formatted text output, exit code indicates success/failure.

Streaming: Line-by-line output for streams.

SDK Library Client

Programming language library for applications:

Request: Function call with typed parameters.

Mapping: Function name maps to operation.

Execution: Library serializes request, sends to platform, deserializes response.

Response: Typed result objects.

Streaming: Iterator or observable pattern for streams.

API Discovery

Clients can discover available operations:

List Operations: API endpoint returns available operations with descriptions.

Operation Schema: Each operation provides input/output schema.

Examples: Operations include usage examples.

Documentation: API provides links to detailed documentation.

Discovery enables dynamic clients that adapt to platform capabilities.

Rate Limiting and Quotas

Platforms may enforce limits:

Rate Limits: Maximum requests per time period (per client, per operation type).

Quotas: Maximum resource consumption (storage, compute, emissions).

Throttling: Slowing requests when approaching limits.

Clients receive feedback about limits:

- Current usage
- Remaining quota
- Reset time
- Throttle status

Caching and ETags

For read operations, caching improves performance:

Resource ETags: CID serves as natural ETag (content-based).

Conditional Requests: "If-None-Match: CID" returns 304 Not Modified if unchanged.

Cache Headers: Platform provides cache duration hints.

Content addressing makes caching reliable—CID guarantees content identity.

Bulk Operations

For efficiency, API supports bulk operations:

Batch Submission: Submit multiple resources in one request.

Batch Retrieval: Retrieve multiple resources by CID list.

Batch Operations: Execute multiple operations atomically.

Bulk operations reduce round-trips and enable atomic multi-resource transactions.

WebSocket/Streaming Support

For real-time interaction:

Persistent Connection: WebSocket or similar for bidirectional communication.

Operation Streaming: Submit operations, receive results over same connection.

Emission Subscription: Subscribe to emission streams, receive items as emitted.

Multiplexing: Multiple operations/streams over one connection.

Streaming enables responsive interactive applications.

API Observability

The API provides visibility into platform behavior:

Operation Logs: Record of API operations performed.

Performance Metrics: Latency, throughput, error rates per operation.

Resource Metrics: Store size, emission rate, projection frequency.

Client Metrics: Per-client usage patterns.

Observability helps clients understand platform health and optimize usage.

Next Steps

The platform API enables clients to interact with the system, but the platform itself must bootstrap from minimal primitives. The next section, Part IV: System Architecture, begins with **Bootstrap Architecture**—how the platform initializes itself from a minimal content-addressed store to a fully functional projection-emission system.

Bootstrap Architecture

From Minimal Store to Self-Describing Platform

The Hologram platform is **self-describing**—it contains its own definition as projections in the store. But this creates a bootstrapping challenge: how does the platform initialize when it needs projections to understand projections?

This document describes the bootstrap sequence—how the platform cold-starts from minimal primitives and builds up to full self-describing operation.

The Bootstrap Problem

The platform requires:

- Projection definitions to understand how to project
- Container store to hold projection definitions
- Projection engine to execute projection definitions

But:

- Projection definitions are resources in the store
- The engine needs projection definitions to project resources
- Projecting projection definitions requires the engine

This is the **bootstrap paradox**: the system needs itself to start itself.

Bootstrap Solution: Staged Initialization

The solution is **staged initialization**—starting with minimal hard-coded capabilities and progressively replacing them with projected capabilities.

Each stage builds on the previous, until the platform is fully self-describing and no hard-coded logic remains (or only fundamental primitives remain).

Bootstrap Layers

Layer 0: Minimal Content-Addressed Store

What Exists: A content-addressed storage implementation providing the four core capabilities:

- Store (write resource, get CID)
- Retrieve (read resource by CID)
- Reference (extract CIDs from resource)
- Query (identify resources by basic criteria)

What's Hard-Coded: The store implementation itself (filesystem, database, memory).

What's Not Present: Projection engine, projection definitions, container types, operations.

This is the **axiom layer**—the minimal foundation assumed to exist.

Layer 1: Store Definition (hologram.store)

What Happens: Emit the store's own definition as a resource.

Resource Content: JSON document describing:

- Store capabilities (store, retrieve, reference, query)
- Store properties (immutability, content addressing, deduplication)
- Store interface (how to interact with store)

Process:

1. Hard-coded bootstrap code creates store definition resource
2. Computes CID
3. Stores resource in the store
4. The store now contains its own definition

Result: The store is **introspectable**—its definition exists as a resource.

What's Still Hard-Coded: Projection engine (doesn't exist yet).

Layer 2: Projection Language (hologram.projection)

What Happens: Emit the projection language definition as resources.

Resource Content:

- Projection definition schema (what fields projection definitions have)
- Conformance requirement schema (how to specify conformance)
- Query language schema (how to express queries)
- Transformation rule schema (how to define transformations)

Process:

1. Hard-coded bootstrap code creates projection language definition
2. Stores as resources in the store
3. Hard-coded minimal projection engine loads these definitions
4. Engine can now interpret projection definitions

Result: The projection language is **self-defined**—its schema exists as resources.

What's Still Hard-Coded: Minimal projection engine (can interpret but is hard-coded).

Layer 3: Base Container Schemas

What Happens: Emit schemas for base container types.

Resource Content:

- hologram.component schema
- hologram.instance schema
- hologram.interface schema
- hologram.documentation schema
- hologram.test schema
- hologram.manager schema
- hologram.view schema

- hologram.dependency schema
- hologram.build schema
- hologram.log schema

Process:

1. Bootstrap code creates JSON Schema definitions for each container type
2. Stores schemas as resources
3. Engine can now validate resources against these schemas

Result: Container types are **schema-defined** rather than hard-coded.

What's Still Hard-Coded: Operation implementations (projection engine can validate but not execute operations).

Layer 4: Projection Definitions

What Happens: Emit projection definitions for each container type.

Resource Content: For each container type, a projection definition specifying:

- Query to identify resources
- Conformance requirements (what resources must be present)
- Aggregation rules (how to follow references)
- Transformation rules (how to structure container)
- Schema references (what schemas apply)

Process:

1. Bootstrap code creates projection definitions using projection language from Layer 2
2. Stores projection definitions as resources
3. Engine loads projection definitions
4. Engine can now project resources into containers per definitions

Result: Container types are **projection-defined** rather than hard-coded.

What's Still Hard-Coded: Basic projection engine logic (query evaluation, reference traversal, transformation application).

Layer 5: Engine Definition (hologram.engine)

What Happens: Emit the projection engine's own definition as resources.

Resource Content:

- Engine capabilities (query execution, traversal, transformation)
- Engine algorithm descriptions (how it processes projections)
- Engine extension points (how to add custom query types, validators, transformations)

Process:

1. Bootstrap code creates engine definition
2. Stores as resources in the store
3. Advanced implementations could use this to build engines (compilation, optimization)

Result: The engine is **self-described**—its behavior is documented in resources.

What's Still Hard-Coded: Actual engine implementation (but its behavior is defined in resources).

Layer 6: Operation Definitions

What Happens: Emit operation definitions as projection definitions.

Resource Content: For each operation (component.create, instance.start, etc.):

- Input schema
- Projection phase specification
- Execution logic description
- Emission pattern
- Output schema

Process:

1. Bootstrap code creates operation definitions
2. Stores as resources
3. Platform can now execute operations per definitions

Result: Operations are **definition-driven** rather than hard-coded.

What's Still Hard-Coded: Execution logic for operations (projections define what to do, but implementation executes it).

Layer 7: Complete Self-Description

What Happens: The platform contains complete definition of itself.

What's in Store:

- Store definition
- Projection language definition
- All container type schemas and projection definitions
- Engine definition
- Operation definitions
- Platform API definition
- Bootstrap process definition (this document as a resource!)

Result: The platform is **fully self-describing**—everything about its behavior exists as resources.

What's Still Hard-Coded: Only Layer 0 (minimal CAS store) and basic projection engine execution. Everything else is defined in resources.

Bootstrap Sequence

The actual initialization sequence when starting a Hologram platform:

Step 1: Initialize Store

Start with empty or existing content-addressed store.

If empty: create minimal store implementation.

If existing: load store and check for bootstrap resources.

Step 2: Check Bootstrap State

Query store for bootstrap marker resource.

If not present: platform is uninitialized, proceed with bootstrap.

If present: platform is already bootstrapped, load existing definitions.

Step 3: Emit Foundation Resources (if bootstrapping)

Execute Layer 0-2 bootstrap:

1. Emit hologram.store definition
2. Emit hologram.projection definition
3. Emit projection language schemas

Step 4: Initialize Minimal Engine

Create minimal projection engine with hard-coded logic:

- Basic query evaluation
- Reference traversal
- Schema validation
- Simple transformations

Engine loads projection language definitions from store.

Step 5: Emit Container Definitions

Execute Layer 3-4 bootstrap:

1. Emit base container type schemas
2. Emit projection definitions for each container type

Engine can now project these container types.

Step 6: Emit Operation Definitions

Execute Layer 6 bootstrap:

1. Emit operation definitions for all platform operations

Platform can now execute operations per definitions.

Step 7: Emit Bootstrap Marker

Create bootstrap marker resource indicating:

- Bootstrap completion timestamp
- Platform version
- Definitions emitted

Emit marker to store.

Step 8: Verify Bootstrap

Execute validation projection:

1. Project all bootstrap resources
2. Verify schemas valid
3. Verify projection definitions valid
4. Verify operations defined

If validation passes: bootstrap complete.

If validation fails: abort, report errors.

Step 9: Enter Normal Operation

Platform is now fully bootstrapped and operational.

All operations use projection-emission cycle.

Platform is self-describing and can introspect its own definitions.

Bootstrap Resources

The bootstrap resources are the **platform kernel**—minimal set of resources required for self-describing operation.

These resources form the **base projection set** that all other projections build upon.

Bootstrap resources should be:

- Minimal (only essential definitions)
- Stable (rarely change, only for platform evolution)
- Well-documented (critical to platform understanding)
- Versioned (enable platform upgrades)

Incremental Bootstrap

For large platforms, bootstrap can be incremental:

Phase 1: Core projection system (Layers 0-4).

Phase 2: Base operations (component CRUD, instance lifecycle).

Phase 3: Extended container types (dependency, build, log).

Phase 4: Views and optimization (catalog, materialized views).

Phase 5: Advanced features (distributed execution, federation).

Each phase emits resources, expands platform capabilities, enables next phase.

Bootstrap from Import

Instead of hard-coded bootstrap, platform can bootstrap from resource import:

Step 1: Start with minimal store + engine.

Step 2: Import bootstrap resources from external source:

- File bundle
- Remote repository
- Another Hologram instance

Step 3: Engine loads imported definitions.

Step 4: Platform operational.

This enables platform distribution as resource bundles.

Bootstrap Verification

After bootstrap, verify platform consistency:

Definition Consistency: All projection definitions reference valid schemas.

Schema Validity: All schemas are well-formed JSON Schema.

Reference Integrity: All CID references in bootstrap resources exist.

Operation Completeness: All required operations are defined.

Engine Capability: Engine can execute all projection types defined.

Verification ensures bootstrap produced valid platform state.

Evolution After Bootstrap

Once bootstrapped, the platform evolves through normal projection-emission:

New Container Types: Emit new projection definitions → new container types available.

New Operations: Emit new operation definitions → new operations executable.

Improved Schemas: Emit updated schemas → refined validation.

Platform Updates: Emit updated bootstrap resources → platform evolves.

Evolution uses the same mechanisms as application development.

The Minimal Kernel

What must remain hard-coded (cannot be bootstrapped away)?

Content-Addressed Storage: The fundamental primitive (store, retrieve, reference, query).

Projection Execution: The ability to evaluate projections (though projection definitions can describe how).

Resource Serialization: Converting between in-memory and byte representations.

These form the **irreducible kernel**—the minimal hard-coded logic required.

Everything else can be defined as resources through projections.

Bootstrap Reproducibility

Bootstrap should be reproducible:

- Same bootstrap code produces same resources
- Same resources produce same CIDs
- Same CIDs produce identical platform state

Reproducibility enables:

- Deterministic platform creation
- Verification of platform integrity
- Consistency across distributed instances

Next Steps

With bootstrap complete, the platform exhibits key properties that make it robust and evolvable. The next document, **Platform Properties**, describes the guarantees, characteristics, and emergent properties that result from the projection-emission model.

Platform Properties

Guarantees and Emergent Characteristics

The Hologram platform exhibits fundamental properties that emerge from its projection-emission architecture. These properties provide guarantees about platform behavior and enable powerful capabilities.

This document catalogs the platform's key properties—both guaranteed invariants and emergent characteristics.

Immutability

Property

Resources in the container store **never change** after emission. Once a resource is stored with a CID, that CID always refers to the same content.

Guarantees

Stable References: A CID reference ispermanent and reliable—it will always retrieve the same content.

Historical Completeness: All versions of evolving data persist as distinct resources. No information is lost through updates.

Concurrent Safety: Multiple readers can access the same resource without coordination—it will never change under them.

Verifiable Integrity: Retrieved content can be hashed and verified against the CID cryptographically.

Implications

Versioning is Natural: Updates create new resources with new CIDs, automatically creating version history.

Caching is Simple: Immutable resources can be cached indefinitely without invalidation concerns.

Distribution is Safe: Resources can be copied between stores without synchronization—content identity is preserved.

Audit Trails are Built-In: Complete operation history exists as immutable resources.

Content Addressing

Property

Resources are identified by **cryptographic hash of their content**, not by location or assigned identifiers.

Guarantees

Identity from Content: Identical content produces identical CID regardless of when, where, or by whom it was created.

Collision Resistance: Different content produces different CIDs with cryptographic probability.

Deduplication: Identical content stored multiple times occupies space once.

Location Independence: CID is valid across stores, networks, time—content can move freely.

Implications

Global Namespace: CIDs are globally unique without coordination or central registry.

Trustless Verification: Content authenticity verifiable without trusting the source.

Efficient Storage: No redundant copies of shared resources (common dependencies, standard libraries).

Portable References: CID references work across platform instances and implementations.

Projection Purity

Property

Projections are **pure queries** that do not modify the store or produce side effects.

Guarantees

Determinism: Same store state and parameters always produce same projection result.

Repeatability: Projections can be executed multiple times without affecting results.

Concurrency: Multiple projections execute safely in parallel without interference.

Reproducibility: Historical projections can be re-executed against historical store state.

Implications

Testing is Straightforward: Projections testable with known resource sets and expected results.

Debugging is Possible: Projection failures reproducible with same inputs.

Optimization is Safe: Caching, parallelization, reordering don’t affect semantics.

Temporal Queries Work: Projecting historical state produces accurate historical views.

Emission Atomicity

Property

Emissions are **atomic operations**—resources are either fully stored and indexed, or not stored at all.

Guarantees

No Partial Writes: Resources are complete and valid or don't exist.

Consistent State: Store remains consistent regardless of emission success or failure.

Transaction Support: Multiple related emissions succeed or fail together.

Recovery from Failure: Failed emissions leave no partial state requiring cleanup.

Implications

Operations are Reliable: Operations either complete successfully or fail cleanly.

Concurrent Emissions Safe: Multiple clients emitting concurrently maintain consistency.

Error Recovery Simple: Failed operations can be retried without cleanup.

Multi-Resource Consistency: Component definitions with multiple resources appear atomically.

Extensibility

Property

New container types, operations, and capabilities can be added by **emitting new projection definitions**.

Guarantees

No Platform Modification Required: Extension through resource emission, not code changes.

Backward Compatibility: Existing projections continue working when new projections added.

Composition Supported: New projections can compose existing projections.

Version Coexistence: Multiple versions of projections can exist simultaneously.

Implications

Platform Evolves Continuously: New capabilities added without platform downtime.

Domain-Specific Extensions: Applications can define custom container types for their needs.

Experimentation is Safe: New projections can be tested without affecting existing system.

Migration is Gradual: Old and new projection versions coexist during transitions.

Self-Description

Property

The platform **contains its own definition** as resources in the store.

Guarantees

Introspectable: Platform behavior is documented in resources, accessible via projections.

Evolvable: Platform definition can be updated by emitting new definition resources.

Bootstrappable: New platform instances can initialize from definition resources.

Documentable: Platform documentation exists as resources, version-controlled with platform.

Implications

Understanding is Accessible: Projecting platform definitions reveals how platform works.

Meta-Operations Possible: Operations that analyze or transform platform definitions.

Migration is Defined: Platform upgrades are resource updates, trackable and reversible.

Consistency Verifiable: Platform definition consistency checkable via projections.

Auditability

Property

All operations leave **immutable audit trails** as emitted resources.

Guarantees

Complete History: Every operation, emission, and state change recorded.

Tamper-Evident: Immutable resources and content addressing prevent undetectable modification.

Traceable: Resource reference graph shows relationships and provenance.

Queryable: Audit data is projectable like any resources.

Implications

Compliance Enabled: Regulatory audit requirements satisfiable from store content.

Debugging Informed: Complete history available for investigating issues.

Attribution Clear: Who emitted what resources when is recorded.

Reproducibility Supported: Historical operations reproducible from audit data.

Composability

Property

Projections, operations, and containers **compose cleanly** without tight coupling.

Guarantees

Hierarchical Composition: Projections can project other projections.

Operation Chaining: Operations compose into workflows.

Container Reuse: Containers participate in multiple projections.

Independent Evolution: Composed elements evolve independently.

Implications

Complex from Simple: Sophisticated capabilities built from basic projections.

Reusability High: Components, projections, operations reused across contexts.

Coupling Low: Changes to one projection don’t require changes to others.

Modularity Maintained: System organized as composable modules, not monolith.

Consistency Models

Property

The platform supports **multiple consistency models** appropriate to different use cases.

Guarantees

Strong Consistency Available: Operations can require immediate consistency when critical.

Eventual Consistency Supported: Operations can accept eventual consistency for performance.

Snapshot Consistency Provided: Projections can use consistent historical snapshots.

Client Choice: Consistency level selected per operation.

Implications

Flexibility: Applications choose appropriate tradeoffs between consistency and performance.

Scalability: Eventual consistency enables distributed, high-throughput scenarios.

Correctness: Strong consistency ensures critical operations maintain invariants.

Optimization: Views and caches use eventual consistency without compromising safety.

Distribution

Property

Resources and projections are **location-independent** and distributable.

Guarantees

CID Portability: CIDs valid across distributed stores.

Content Synchronization: Resources copyable between stores with identity preserved.

Projection Mobility: Projections executable on any store containing required resources.

Decentralization Possible: No required central coordinator or master.

Implications

Federation Enabled: Multiple Hologram instances can federate, sharing resources.

Edge Computing Supported: Projections executable at edge with local resource subset.

Disaster Recovery Simple: Stores replicable for redundancy and recovery.

Geographic Distribution: Resources locatable near users for performance.

Performance Characteristics

Property

The platform exhibits predictable **performance characteristics** based on resource access patterns.

Guarantees

Content Retrieval O(1): Direct CID retrieval is constant time (hash table lookup).

Deduplication Automatic: Storage scales with unique content, not total references.

Projection Cost Proportional: Projection cost scales with resources accessed, not store size.

Caching Effective: Immutability enables aggressive caching without invalidation complexity.

Implications

Scalability Predictable: Performance behavior understood and plannable.

Optimization Opportunities: Materialized views, indexes, caching optimize common patterns.

Resource Planning: Storage and compute requirements estimable from usage patterns.

Bottleneck Identification: Performance issues traceable to specific projection or emission patterns.

Backward Compatibility

Property

Platform evolution maintains **backward compatibility** through projection versioning.

Guarantees

Old Projections Work: Existing projection definitions continue functioning after platform updates.

Old Resources Accessible: Historical resources remain retrievable and projectable.

Version Coexistence: Multiple projection versions available simultaneously.

Gradual Migration: Systems transition from old to new projections at their own pace.

Implications

Breaking Changes Avoidable: New projection versions published alongside old versions.

Deprecation Gradual: Old projections marked deprecated but remain functional.

Migration Risk Low: New versions testable before switching production systems.

Long-Term Stability: Systems built on Hologram remain functional through platform evolution.

Resource Efficiency

Property

The platform minimizes **resource consumption** through deduplication and structural sharing.

Guarantees

No Redundant Storage: Identical content stored once regardless of reference count.

Efficient Updates: Updates share unchanged resources, storing only differences.

Lazy Loading: Resources retrieved only when projected, not preemptively.

Garbage Collection: Unreferenced resources reclaimable to free storage.

Implications

Storage Costs Bounded: Storage grows with unique content, deduplicating redundancy.

Network Efficiency: Only missing resources transferred between stores.

Memory Efficiency: Projection engine loads only required resources.

Cost Optimization: Storage and transfer costs minimized through structural sharing.

Temporal Capabilities

Property

Immutability enables **time-travel queries** and historical projections.

Guarantees

Historical State Preserved: All resource versions remain in store.

Temporal Projections Valid: Projections can target specific points in time.

Consistency Across Time: Historical projections produce consistent views of past state.

Provenance Trackable: Resource creation and modification timeline reconstructable.

Implications

Debugging Simplified: Issues reproducible by projecting historical state.

Compliance Supported: Historical compliance queries answerable.

Analytics Enabled: Time-series analysis of platform state and resources.

Undo Possible: Reverting to previous state means projecting historical resources.

Security Properties

Property

Content addressing and immutability provide **security foundations**.

Guarantees

Integrity Verification: Content authenticity cryptographically verifiable.

Tamper Detection: Any modification changes CID, making tampering evident.

Access Control Enforcement: Store can enforce read/write permissions per resource or namespace.

Audit Trail Immutable: Audit resources cannot be altered retroactively.

Implications

Trust Minimized: Verify content cryptographically rather than trusting source.

Compliance Enhanced: Immutable audit trails satisfy regulatory requirements.

Attack Surface Reduced: Immutability eliminates modification attacks.

Provenance Verifiable: Resource origin and modification history checkable.

Next Steps

These properties emerge from the platform's architecture, but realizing them requires implementation decisions. The next document, **Implementation Considerations**, discusses how to implement the platform—architectural patterns, technology choices, performance optimization, and practical tradeoffs.

Implementation Considerations

Architectural Patterns and Practical Tradeoffs

Implementing the Hologram platform requires translating the conceptual model into running systems. This document provides guidance on implementation architecture, technology choices, optimization strategies, and practical tradeoffs—without prescribing specific implementations.

Implementation Principles

Separation of Concerns

Implementations should separate:

Store Backend: Content-addressed storage implementation (filesystem, database, object storage).

Projection Engine: Interprets projection definitions and executes projections.

API Layer: Protocol bindings (HTTP, MCP, CLI) that expose operations to clients.

Operation Logic: Implements execution phase of operations.

Client Libraries: Language-specific wrappers for API consumption.

Clean separation enables:

- Independent evolution of each component
- Multiple backend options without changing engine
- Multiple API protocols without changing operations
- Testing components in isolation

Implementation Independence

The platform model is abstract—multiple valid implementations exist:

Language Choices: TypeScript, Go, Rust, Python, Java, or any language with serialization and hashing.

Store Backends: Filesystem, PostgreSQL, SQLite, MongoDB, S3, IPFS, or custom.

Engine Strategies: Interpreted, compiled, JIT, or distributed execution.

API Protocols: HTTP/REST, gRPC, GraphQL, MCP, or custom protocols.

The model’s semantics remain consistent across implementations.

Store Implementation Patterns

Filesystem Backend

Structure: Content-addressed files in directory hierarchy.

CID to Path Mapping: Typically first N hex digits as subdirectories for distribution (e.g., `ab/cd/abcd123...`).

Advantages:

- Simple implementation
- Works with standard filesystem tools
- Git-compatible (if using predictable formatting)
- Easy backup and replication

Disadvantages:

- Poor query performance (requires scanning)
- Limited concurrent write scalability
- No built-in indexing

Best For: Small to medium stores, development, git-integrated workflows.

Relational Database Backend

Structure: Resources as BLOBs in tables, CIDs as primary keys.

Schema:

- Resources table: (CID, content, content_type, timestamp, size)
- References table: (source_CID, target_CID) for edges
- Metadata tables: indexes for queries

Advantages:

- Efficient queries via SQL
- ACID transactions for atomicity
- Mature tooling and operations
- Scalable with proper indexing

Disadvantages:

- Schema somewhat rigid
- Large BLOBs can stress database
- More complex setup than filesystem

Best For: Medium to large stores, production systems, complex queries.

Object Storage Backend

Structure: Resources as objects in cloud storage (S3, Azure Blob, GCS).

Key Scheme: CID as object key.

Metadata: Object metadata stores content type, timestamps.

Advantages:

- Massive scalability
- High durability and availability
- Cost-effective for large stores
- Geographic distribution

Disadvantages:

- Higher latency than local storage
- Query requires external index
- Cost per API call
- Network dependency

Best For: Large-scale, distributed, cloud-native deployments.

Hybrid Architectures

Local + Remote: Local filesystem cache, remote object storage for persistence.

Database + Object Storage: Database for metadata and small resources, object storage for large artifacts.

Tiered Storage: Hot data in fast storage (SSD, database), cold data in cheap storage (object storage, archive).

Hybrid architectures balance performance, scalability, and cost.

Indexing Strategies

Content Indexing

For efficient queries, index resource content:

Full-Text Search: Index textual content for search queries (Elasticsearch, PostgreSQL FTS).

Field Indexing: Extract and index specific JSON fields (namespace, version, tags).

Spatial Indexing: For geographic or geometric data (PostGIS).

Indexing trades storage and indexing cost for query performance.

Reference Indexing

Index the resource reference graph:

Forward References: Given a resource, what does it reference (adjacency list).

Reverse References: What resources reference this one (reverse index).

Graph Database: Specialized graph databases (Neo4j) for complex graph queries.

Reference indexes enable efficient dependency resolution and graph traversal.

Metadata Indexing

Index store metadata:

Timestamp Indexes: Query resources by emission time.

Size Indexes: Find large resources, compute storage statistics.

Access Indexes: Track access patterns for cache optimization.

Metadata indexes support operations beyond content queries.

Interpreter Pattern

Engine interprets projection definitions at runtime:

Advantages:

- Simple implementation
- No compilation step
- Dynamic projection definitions

Disadvantages:

- Slower execution than compiled
- Less optimization opportunity

Best For: Prototypes, small scales, frequently changing projections.

Compiler Pattern

Engine compiles projection definitions to native code:

Advantages:

- Faster execution
- Optimization opportunities (inlining, loop unrolling)
- Better resource utilization

Disadvantages:

- Complex implementation
- Compilation overhead
- Requires compilation infrastructure

Best For: Production systems, performance-critical, stable projections.

Hybrid Pattern

Interpret initially, compile hot projections:

Advantages:

- Fast startup (no compilation wait)
- Optimized steady-state (compiled hot paths)
- Adaptive to workload

Disadvantages:

- Most complex implementation
- Profiling and monitoring overhead

Best For: Large-scale production with varied workloads.

Caching Strategies

Resource Content Caching

Cache retrieved resource content:

LRU Cache: Evict least recently used resources when cache full.

Size-Aware Cache: Evict based on resource size and access patterns.

Tiered Cache: Memory cache for hot resources, disk cache for warm resources.

Content caching dramatically reduces store access for repeated retrievals.

Projection Result Caching

Cache projected containers:

Key: (Projection definition CID, parameters, store version).

Invalidation: Invalidate when underlying resources change.

TTL: Time-to-live for eventual consistency.

Result caching avoids re-executing expensive projections.

Index Caching

Cache query results and indexes:

Query Result Cache: Cache results of common queries.

Materialized Views: Pre-compute and cache complex projections as view resources.

Index caching optimizes read-heavy workloads.

Concurrency and Parallelism

Concurrent Projections

Multiple projections can execute concurrently:

Read Parallelism: Projections are pure queries, safe to parallelize.

Resource Pooling: Share resource retrieval across concurrent projections.

Batch Optimization: Group resource retrievals from concurrent projections.

Concurrent projection execution scales with available cores.

Concurrent Emissions

Multiple emissions require coordination:

Optimistic Concurrency: Emit independently, rely on content addressing for deduplication.

Transactional Emissions: Use store backend transactions for atomic multi-resource emissions.

Partition by Namespace: Partition store by namespace for independent emission streams.

Emission concurrency balances consistency and throughput.

Distributed Execution

For large-scale systems, distribute work:

Partition Store: Distribute resources across nodes by CID range.

Projection Routing: Route projection requests to nodes holding required resources.

Result Aggregation: Gather partial results from multiple nodes.

Distribution enables horizontal scalability beyond single-node limits.

Performance Optimization

Query Optimization

Optimize projection query evaluation:

Index Selection: Use indexes for selective queries.

Query Planning: Analyze projection definition, choose optimal execution plan.

Predicate Pushdown: Evaluate filters early to reduce data scanned.

Query optimization is critical for large stores with complex projections.

Traversal Optimization

Optimize reference traversal:

Breadth-First vs Depth-First: Choose based on projection pattern and cache characteristics.

Lazy Loading: Retrieve referenced resources only when needed.

Prefetching: Predict and prefetch likely-needed resources.

Parallel Traversal: Follow multiple references concurrently.

Traversal optimization reduces latency for deep or wide reference graphs.

Serialization Optimization

Optimize resource serialization/deserialization:

Binary Formats: Use efficient binary formats (Protocol Buffers, MessagePack) over JSON where appropriate.

Compression: Compress resources at rest and in transit.

Streaming: Stream large resources rather than loading entirely into memory.

Serialization efficiency impacts both storage and transmission costs.

Scalability Patterns

Vertical Scaling

Single-node optimization:

Memory: More RAM for larger caches, holding more resources.

CPU: More cores for concurrent projection execution.

Storage: Faster storage (NVMe SSD) for quicker resource access.

Vertical scaling is simpler but has hard limits.

Horizontal Scaling

Multi-node distribution:

Replication: Replicate entire store across nodes (read scaling).

Sharding: Partition store across nodes (write scaling, storage scaling).

Load Balancing: Distribute requests across nodes.

Horizontal scaling enables near-unlimited capacity but increases complexity.

Caching Tiers

Multi-level caching:

L1: In-Process Memory: Fastest, smallest, per-node.

L2: Distributed Cache: Fast, larger, shared (Redis, Memcached).

L3: Store Backend: Slower, largest, durable.

Tiered caching balances speed, capacity, and cost.

Error Handling and Resilience

Failure Modes

Handle common failures:

Store Unavailable: Retry with exponential backoff, fail request if timeout exceeded.

Resource Not Found: Return clear error, suggest checking CID or dependencies.

Validation Failure: Return detailed validation errors for fixing.

Timeout: For long operations, support cancellation and resume.

Graceful failure handling improves user experience.

Recovery Strategies

Recover from failures:

Idempotent Retries: Operations are idempotent, safe to retry.

Transaction Rollback: Roll back partial emissions on failure.

Checkpointing: For long operations, checkpoint progress for resume.

Recovery strategies enable robust operations despite failures.

Consistency Maintenance

Ensure store consistency:

Validation on Emit: Validate resources before emission, reject invalid.

Reference Checking: Ensure referenced CIDs exist (or defer to projection time).

Garbage Collection: Periodically remove unreferenced resources to reclaim space.

Consistency maintenance prevents store corruption.

Monitoring and Observability

Metrics

Track platform health:

Store Metrics: Size, growth rate, resource count, retrieval latency.

Projection Metrics: Execution count, latency, cache hit rate, failure rate.

Emission Metrics: Emission rate, validation failure rate, transaction rollback rate.

API Metrics: Request rate, latency, error rate per operation.

Metrics enable proactive issue detection and capacity planning.

Tracing

Trace requests through system:

Distributed Tracing: Track projection execution across components (OpenTelemetry).

Resource Access Tracing: Record which resources are accessed during projections.

Emission Tracing: Track emission flows from operation to store.

Tracing enables debugging complex distributed operations.

Logging

Log platform activity:

Operation Logs: Record operations executed, parameters, results.

Error Logs: Detailed error information for troubleshooting.

Audit Logs: Security-relevant events (authentication, authorization, sensitive operations).

Structured logging enables searching and analysis.

Security Implementation

Authentication

Verify client identity:

API Keys: Simple, suitable for service-to-service.

OAuth/OIDC: Standard for user authentication.

Mutual TLS: Certificate-based for high-security scenarios.

Authentication establishes who is making requests.

Authorization

Enforce access control:

RBAC: Role-based access control (roles grant permissions).

ABAC: Attribute-based access control (context-dependent permissions).

Resource-Level ACLs: Permissions per resource or namespace.

Authorization determines what authenticated clients can do.

Encryption

Protect data:

At Rest: Encrypt resources in store backend.

In Transit: TLS for all network communication.

End-to-End: Clients encrypt before emission, decrypt after projection (for sensitive data).

Encryption protects confidentiality.

Testing Strategies

Unit Testing

Test components in isolation:

Store Interface: Test CRUD operations, CID generation, deduplication.

Projection Engine: Test query evaluation, traversal, transformation with mock store.

Operations: Test operation logic with mock projections and emissions.

Unit tests validate individual components.

Integration Testing

Test components together:

Operation End-to-End: Submit artifacts, create component, validate, retrieve.

Projection Composition: Test nested and sequential projections.

Failure Scenarios: Test error handling, recovery, rollback.

Integration tests validate component interactions.

Performance Testing

Measure performance characteristics:

Load Testing: Measure throughput and latency under load.

Stress Testing: Find breaking points and failure modes.

Scalability Testing: Verify horizontal and vertical scaling behavior.

Performance tests validate scalability and identify bottlenecks.

Deployment Patterns

Single-Node Deployment

Simple deployment:

Components: Store backend, engine, API server on one node.

Suitable For: Development, small deployments, proof-of-concept.

Advantages: Simple, low cost, easy to manage.

Disadvantages: Limited scale, single point of failure.

Distributed Deployment

Multi-node deployment:

Components: Store backend distributed/replicated, multiple engine nodes, load balancer.

Suitable For: Production, large scale, high availability.

Advantages: Scalable, resilient, high performance.

Disadvantages: Complex, higher cost, requires orchestration.

Cloud-Native Deployment

Containerized, orchestrated deployment:

Technologies: Docker containers, Kubernetes orchestration, cloud-managed storage.

Suitable For: Cloud environments, microservices architectures, elastic scaling.

Advantages: Portable, scalable, declarative configuration.

Disadvantages: Complexity, cloud dependency, learning curve.

Next Steps

This document provides implementation guidance without prescribing specific technologies. The final document, **Terminology**, provides a glossary of terms used throughout the documentation and maps Hologram concepts to established computer science terminology.

Terminology

Glossary and Cross-References

This document defines key terms used throughout the Hologram documentation and maps them to established computer science concepts.

Core Concepts

Container Store

The universal storage system where all resources exist as immutable, content-addressed data.

Related CS Concepts: Content-addressable storage (CAS), content-addressable memory, immutable data stores, append-only logs.

See: Document 02 - The Container Store

Resource

Any sequence of bytes stored in the container store. Resources have no inherent type or structure—meaning emerges through projection.

Related CS Concepts: Blob (Binary Large Object), untyped data, raw data, resource (REST).

See: Document 02 - The Container Store

Content Identifier (CID)

A cryptographic hash of a resource’s content, used as the resource’s identifier. Identical content produces identical CIDs.

Related CS Concepts: Content hash, cryptographic hash, checksum, digest, content addressing, self-certifying identifier.

See: Document 02 - The Container Store

Immutability

Resources never change after emission. Once stored, a resource’s content is permanent. Updates create new resources with new CIDs.

Related CS Concepts: Immutable data structures, persistent data structures, write-once storage, append-only.

See: Document 02 - The Container Store, Document 13 - Platform Properties

Projection Concepts

Projection

A query-based definition that identifies, aggregates, and transforms resources into a container. Projections are pure queries without side effects.

Related CS Concepts: View (database), query, transformation, lens (functional programming), map/reduce, query language.

See: Document 04 - Container Projections

Container

A structured collection of resources produced by projecting resources from the store. Containers have defined purpose and relationships.

Related CS Concepts: View (database), aggregate (DDD), composite object, container (general programming).

See: Document 04 - Container Projections

Projection Definition

A resource that specifies how to project resources into containers, including query criteria, aggregation rules, and transformations.

Related CS Concepts: View definition, query definition, schema, specification, declarative program.

See: Document 06 - The Projection Language

Conformance Requirements

Instructions within projection definitions specifying what resources must be present and how they should be structured. Conformance defines projection, not just validation.

Related CS Concepts: Constraints, schema, type specification, contract, invariant.

See: Document 06 - The Projection Language

Projection Engine

The component that interprets projection definitions and executes them against the container store to produce containers.

Related CS Concepts: Query engine, interpreter, virtual machine, execution engine, runtime.

See: Document 05 - The Projection Engine

Emission Concepts

Emission

The process of writing execution results back to the container store as new resources. Emissions make ephemeral results durable.

Related CS Concepts: Write operation, persistence, materialization, output, side effect (though emissions are structured).

See: Document 07 - The Emission Model

Projection-Emission Cycle

The fundamental operating principle: Project resources → Execute operations → Emit results → Store resources → Repeat.

Related CS Concepts: Read-compute-write cycle, ETL (Extract-Transform-Load), data pipeline, event loop.

See: Document 03 - The Projection-Emission Cycle

Emission Stream

A sequence of emissions produced continuously over time, such as logs or state snapshots from a running instance.

Related CS Concepts: Event stream, log stream, observable, reactive stream, data stream.

See: Document 07 - The Emission Model

Container Types

Component Container

A container defining a reusable capability through specifications, interfaces, documentation, tests, and dependencies.

Related CS Concepts: Module, package, library, service definition, component (software engineering).

See: Document 08 - Container Types

Instance Container

A container representing a running execution of a component definition with runtime state.

Related CS Concepts: Process, object instance, runtime instance, container instance (Docker), actor (actor model).

See: Document 08 - Container Types

Interface Container

A container specifying contracts between components through method signatures and type definitions.

Related CS Concepts: Interface (OOP), API specification, contract, protocol, abstract base class.

See: Document 08 - Container Types

View Container

A materialized projection stored as a resource for efficient repeated access.

Related CS Concepts: Materialized view (database), cached result, denormalized data, index, summary table.

See: Document 08 - Container Types, Document 09 - Platform Views

Documentation Container

A container providing human-readable explanations, guides, and examples.

Related CS Concepts: Documentation, readme, manual, help text, javadoc/docstring.

See: Document 08 - Container Types

Test Container

A container defining validation logic for verifying component behavior.

Related CS Concepts: Test suite, test case, test spec, unit test, integration test.

See: Document 08 - Container Types

Platform Architecture

Bootstrap

The process of initializing the platform from minimal primitives to a fully self-describing system.

Related CS Concepts: Bootstrapping, cold start, system initialization, self-hosting compiler.

See: Document 12 - Bootstrap Architecture

Operation

A named procedure that projects resources, executes logic, and emits results. Operations are the platform’s verbs.

Related CS Concepts: Command (CQRS), action, method, procedure, RPC call, API endpoint.

See: Document 10 - Platform Operations

Platform API

The client interface for interacting with the platform—submitting operations, retrieving resources, subscribing to emissions.

Related CS Concepts: API, SDK, client library, REST API, RPC interface.

See: Document 11 - Platform API

Advanced Concepts

Self-Description

The property that the platform contains its own definition as resources in the store. The platform can introspect and evolve itself.

Related CS Concepts: Reflection, introspection, self-hosting, metacircular evaluation, reflective architecture.

See: Document 13 - Platform Properties

Meta-Projection

A projection that projects other projection definitions. Projections analyzing or transforming projections.

Related CS Concepts: Meta-programming, reflection, higher-order function, metaobject protocol.

See: Document 05 - The Projection Engine

Temporal Projection

A projection targeting resources as they existed at a specific point in time. Time-travel queries.

Related CS Concepts: Temporal database, time-travel query, historical query, version control, snapshot.

See: Document 04 - Container Projections

Reference Graph

The directed graph formed by CID references between resources. Nodes are resources, edges are references.

Related CS Concepts: Directed graph, object graph, reference graph, dependency graph, DAG (if acyclic).

See: Document 02 - The Container Store

Garbage Collection

The process of identifying and removing unreferenced resources to reclaim storage space.

Related CS Concepts: Garbage collection, reference counting, mark-and-sweep, reachability analysis.

See: Document 02 - The Container Store

Established CS Mappings

Hologram → Database Systems

- **Container Store** ↔ Database
- **Resource** ↔ Row/Document
- **CID** ↔ Primary Key (content-derived)
- **Projection** ↔ Query/View
- **Emission** ↔ INSERT/UPDATE
- **View** ↔ Materialized View
- **Reference Graph** ↔ Foreign Keys/Relationships

Hologram → Functional Programming

- **Projection** ↔ Pure Function
- **Container** ↔ Product Type/Record
- **Projection Definition** ↔ Function Definition
- **Projection Composition** ↔ Function Composition
- **Immutability** ↔ Immutable Data Structures
- **Conformance** ↔ Type Constraint

Hologram → Content-Addressed Storage

- **Container Store** ↔ CAS (Git, IPFS, Merkle DAG)
- **CID** ↔ SHA hash, Content Address
- **Resource** ↔ Object, Blob
- **Reference** ↔ Pointer, Link
- **Immutability** ↔ Content Addressing Property

Hologram → Version Control

- **Emission** ↔ Commit
- **CID** ↔ Commit Hash
- **Reference Graph** ↔ Commit Graph
- **View** ↔ Working Directory
- **Temporal Projection** ↔ Checkout Historical Commit

Hologram → Container Orchestration

- **Component** ↔ Container Image
- **Instance** ↔ Running Container
- **Manager** ↔ Orchestrator (Kubernetes)
- **Operation** ↔ Controller Action
- **Platform** ↔ Container Runtime

Hologram → Object-Oriented Programming

- **Component** ↔ Class
- **Instance** ↔ Object Instance
- **Interface** ↔ Interface/Abstract Class
- **Operation** ↔ Method
- **Conformance** ↔ Type Constraint/Contract

Document Cross-References

Foundation Concepts

- Container Store: Document 02
- Projection-Emission Cycle: Document 03
- Container Projections: Document 04

Engine and Language

- Projection Engine: Document 05
- Projection Language: Document 06
- Emission Model: Document 07

Platform Components

- Container Types: Document 08
- Platform Views: Document 09
- Platform Operations: Document 10
- Platform API: Document 11

System Architecture

- Bootstrap: Document 12
- Properties: Document 13
- Implementation: Document 14

Acronyms and Abbreviations

API: Application Programming Interface

CAS: Content-Addressed Storage

CID: Content Identifier

CRUD: Create, Read, Update, Delete

DAG: Directed Acyclic Graph

JSON: JavaScript Object Notation

MCP: Model Context Protocol

REST: Representational State Transfer

SHA: Secure Hash Algorithm

TTL: Time To Live

Conventions

Naming Patterns

hologram.{type}: Platform-provided base container types (hologram.component, hologram.instance).

{namespace}.{name}: Application-defined resources following namespace pattern.

{operation}.{action}: Operation naming (component.create, instance.start).

CID Format

CIDs are typically represented as:

- Prefix: “cid:” (optional, implementation-dependent)
- Hash: Hexadecimal digest of content
- Example: “cid:abc123def456...” or “abc123def456...”

Resource References

Resources reference other resources by embedding CIDs in their content, typically as string values in JSON fields.

Conclusion

This terminology document provides definitions for key concepts and maps them to established computer science concepts. Understanding these mappings helps situate Hologram within the broader context of computing systems while appreciating its unique architectural approach.

The Complete Documentation

This completes the 15-document series on the Hologram Container Engine and Platform:

Part I: Foundation - Documents 1-4 establish the core model from first principles.

Part II: The Engine - Documents 5-7 detail the projection engine and emission mechanisms.

Part III: The Platform - Documents 8-11 describe container types, views, operations, and API.

Part IV: System Architecture - Documents 12-15 cover bootstrap, properties, implementation, and terminology.

Together, these documents define the conceptual model that hologram.spec implementations will realize.