

NCP: Neural Computation Protocol (NCP) v0.2.2

Status: Draft v0.2.2 (non-breaking clarifications: artifact extension point, stochastic external-model semantics, MCP integration appendix, cost metadata)

Compatibility: v0.2.2 is backward-compatible with v0.2 and does not change the normative core execution model.

Versioning Policy

- **v0.2.x series:** MUST remain backward-compatible within the series. Patch releases (x) may add clarifications, appendices, and optional fields, but MUST NOT change normative execution semantics.
- **Breaking changes:** MUST increment the minor version (e.g., v0.3.0).
- **Stability target:** v1.0.0 will designate the first stable major release of the protocol.

0. Summary

NCP (Neural Computation Protocol) standardizes **composable, auditable neural primitives for agentic systems**. A **Brick** is the computation unit; a **Graph** is the composition.

Subtitle (recommended): *A Standard for Composable, Auditable AI Agent Primitives* NCP standardizes **pure-functional, sandboxed computation units** ("Bricks") and **graph manifests** that compose them into an auditable, replayable, scalable network. Intelligence lives in **graph topology + synaptic state**, not in individual Bricks.

NCP defines:

- Brick packaging, identity, versioning, signing, and attestations
- Brick invocation envelope (state, time, graph refs, provenance)
- Typed result + error model with **structural boundaries**
- Carry-state transport (inline vs handle), CAS semantics, and failure-side-effects
- Graph ref capability model via **symbolic slot binding**
- Graph-level routing policies for success + typed errors
- Conformance suites for determinism and sandbox/security profiles

1. Conformance Language

The keywords **MUST**, **MUST NOT**, **SHOULD**, **SHOULD NOT**, and **MAY** are to be interpreted as described in RFC 2119.

2. Goals and Non-goals

2.1 Goals

1. **Purity by construction:** Bricks are pure functions; runtime enforces via sandbox + capability model.
2. **Replayability:** Given recorded inputs, pinned graph version, and recorded time values (where applicable), execution is replayable.
3. **Composability:** Bricks are portable; graphs bind topology-specific details.
4. **Auditability:** Minimal trace records provide verifiable provenance for every invoke.
5. **Scalability:** Stateless compute scaling is preserved; state is explicit and runtime-owned.

2.2 Non-goals

- Standardizing training/learning pipelines (NCP defines serving semantics; learning produces new graph versions).
 - Defining a specific storage engine (NCP defines state semantics; implementations vary).
 - Defining UI/marketplace policies (NCP defines lifecycle metadata that enables them).
-

3. Core Concepts

3.0 Naming

- **NCP** refers to the protocol.
- **Brick** refers to the protocol-compliant computation unit.
- **Graph** refers to the composition of Bricks.

3.1 Brick

A **Brick** is a packaged computation artifact with:

- Input schema, Output schema
- Optional carry-state schema
- Manifest-declared determinism + time model
- Manifest-declared graph-ref slots (capabilities)
- Strict resource limits
- Zero ambient authority

3.2 Graph

A **Graph** composes Bricks into nodes connected by typed edges. The graph binds **symbolic ref slots** to concrete graph refs (weights, thresholds, params).

3.3 State Taxonomy

NCP defines three layers of state:

1. **Execution state**: ephemeral values inside a single invoke
2. **Graph state (runtime-owned)**:
3. **Synaptic state (cold, versioned)**: weights, thresholds, topology params
4. **Carry state (hot, session-scoped)**: activation vectors, windows, counters
5. **Observability state**: append-only trace/audit/log records; write-only from Brick perspective

3.4 Runtime

The **Runtime** executes Bricks inside a sandbox, routes signals per graph policy, and owns all state storage, time injection, and tracing.

4. Identifiers

- `brick_id`: globally unique, namespaced string (e.g., `org.ncp-commons.sentiment-gate`)
 - `brick_version`: SemVer string (e.g., `1.4.2`)
 - `graph_id`: globally unique, namespaced string
 - `graph_version`: immutable identifier (hash or monotonically increasing version)
 - `session_id`: unique per session/trajectory
 - `trace_id`: unique per end-to-end request
-

5. Brick Bundle Format

A Brick is distributed as an immutable **Bundle** containing:

1. **Brick Manifest** (canonicalized for hashing)
2. **Schemas** (input/output/carry-state)
3. **Artifact** (e.g., WASM or ONNX-in-sandbox)
4. **Signature(s)**
5. **Attestation(s)** (optional but strongly recommended; may be required by runtime policy)

5.1 Required Bundle Digests

A bundle MUST provide:

- `artifact_digest`: hash of artifact bytes
- `manifest_digest`: hash of canonical manifest
- `bundle_digest`: hash over (`manifest_digest` + `artifact_digest` + `schema_digests` + `attestation_digests`)

5.2 Canonicalization

Implementations MUST define and use a canonical serialization (e.g., canonical JSON) for digests and signatures to avoid equivalent-but-different encodings.

6. Brick Manifest (Normative)

6.1 Identity and Compatibility

A Brick Manifest MUST include:

- `brick_id`
- `version` (SemVer)
- `nep_protocol_range` (e.g., `>=0.2.0 <0.3.0`)
- `required_runtime_features[]` (may be empty)
- `status`: `active` | `deprecated` | `sunset`
- `successor` (optional)

6.2 Artifact

Manifest MUST include:

- `artifact.format`: `wasm` (v0.2 REQUIRED)
- `artifact.entrypoint`: string (e.g., `invoke`)
- `artifact.digest`
- `artifact.size_bytes`

Future formats (extension point): Future protocol versions MAY define additional `artifact.format` values (e.g., `wasm_aot`, `intrinsic`, `native_sandboxed`).

Runtimes MUST NOT execute non-`wasm` artifacts unless:

1. the format is defined in a ratified NCP protocol version, and
2. the runtime has a conformance-tested sandbox and ABI for that format, and
3. the format satisfies the same purity, determinism, and capability constraints in Sections 6.5–6.7.

Note: If an implementation introduces runtime-provided **intrinsic**s in a future version, it MUST preserve replay/auditability by recording sufficient runtime identity in tracing (e.g., runtime vendor/version and intrinsic implementation digest or equivalent).

6.3 Schemas

Manifest MUST include:

- `schemas.input`
- `schemas.output`

- `schemas.carry_state` (nullable)
- `schemas.metadata` (optional)

6.4 Resource Limits

Manifest MUST include:

- `limits.max_ms`
- `limits.max_mem_mb`
- `limits.max_output_bytes`
- `limits.max_input_bytes` (recommended)

Manifest MAY include (informational metadata):

- `limits.estimated_cost_per_invoke_usd` (number)

Rules:

- `estimated_cost_per_invoke_usd` is NOT enforced by the runtime. Tooling MAY use it for budget modeling and alerts.

6.5 Purity and Capabilities

Bricks MUST run with **zero ambient authority**.

Manifest MUST declare:

- `capabilities[]` (v0.2 MUST be empty)

Runtime MUST enforce:

- no filesystem access
- no network access
- no wall-clock access except via declared time model
- no nondeterministic randomness except via injected seed

6.6 Determinism

Manifest MUST declare:

- `determinism.mode`: `bit_exact` | `bounded_drift` | `stochastic`
- If `bounded_drift`, MUST declare `determinism.epsilon`
- If `stochastic`, MUST declare `determinism.seed_source`: `ctx.seed` (v0.2)

6.6.1 Stochastic Bricks with External Model Dependencies

A Brick that delegates computation to an external model dependency (e.g., an LLM inference endpoint) MUST declare `determinism.mode: stochastic`.

Such Bricks MUST additionally declare:

- `determinism.model_pin`: an identifier sufficient to characterize the model dependency for audit and cache semantics (e.g., provider/model name + version/date; optionally quantization/engine version if applicable).
- `determinism.reproducibility_level`: `seed_deterministic` | `seed_approximate`

Semantics:

- `seed_deterministic`: Given identical input, identical `ctx.seed`, identical `model_pin`, and identical runtime environment, the Brick is expected to produce identical output.
- The runtime MAY cache results for this Brick, and MUST include `model_pin` in the cache key.
- `seed_approximate`: Identical `ctx.seed` does NOT guarantee identical output across invocations (common for third-party APIs and distributed inference services).
- The runtime MUST NOT serve cached results for this Brick.
- The runtime MUST record the full output payload for audit/forensic replay (see Section 11.1), or MUST record a stable reference to an operator-controlled secure audit store that contains the full output.

Runtimes MUST include `model_pin` and `reproducibility_level` in trace records for invocations of such Bricks.

6.7 Time Model

Manifest MUST declare:

- `time_model`: `pure` | `logical` | `wall_bounded`

Cross-validation rules:

- If `time_model=wall_bounded`, determinism MUST NOT be `bit_exact`.
- If `time_model=pure`, runtime MUST NOT inject time fields.
- If `time_model=logical`, runtime MUST inject `ctx.logical_time`.
- If `time_model=wall_bounded`, runtime MUST inject `ctx.wall_time` and MUST record it in trace.

6.8 Carry State Policy

Manifest MUST declare:

- `carry_state_class`: `none` | `tiny` | `medium` | `large`
- `carry_state_transport`: `inline` | `handle`
- `carry_state_max_bytes`

Rules:

- If `schemas.carry_state` is null, `carry_state_class` MUST be `none`.

- If `carry_state_transport=inline`, runtime MUST reject invocations whose `carry_state` exceeds `carry_state_max_bytes`.
- If `carry_state_transport=handle`, runtime MUST provide a `StateHandle` and MAY provide `carry_state_working_set`.

6.9 Failure Side-effects Channel

Manifest MAY declare:

- `carry_state_side_effects_schema` (nullable)

Rules:

- On `Failure`, Brick MUST NOT return `carry_state_next`.
- On `Failure`, Brick MAY return `carry_state_side_effects` **only if** `carry_state_side_effects_schema` is declared.
- Runtime MUST validate `carry_state_side_effects` against the declared schema.

6.10 Graph Ref Slots (Capabilities)

Manifest MUST declare `graph_ref_slots[]`, where each slot includes:

- `slot`: stable symbolic name
- `type`: scalar or structured type (e.g., `float`, `int`, `tensor/f16[16]`)
- `access`: `read`
- `ref_consistency`: `PINNED` | `LIVE_OPS`
(`LIVE_LEARNING` is reserved for a future major version.)

Runtime MUST enforce:

- Brick can only read refs provided for declared slots.
- Any attempt to access undeclared refs MUST produce `RUNTIME_REJECTED`.

7. Graph Manifest (Normative)

7.0 Fan-in and Node Activation

Graphs MUST define **node activation policies** to make fan-in semantics portable across runtimes.

Each node MAY include an `activation` section:

- `mode`: `all_required` | `any` | `quorum`
- `all_required`: runtime MUST wait until all required input fields are satisfied (see required fields below) or `timeout_ms` elapses.
- `any`: runtime MAY invoke when the first qualifying inbound delivery arrives (useful for redundancy).

- `quorum`: runtime MUST invoke when `quorum_n` of `quorum_m` inbound deliveries arrive.
- `timeout_ms`: non-negative integer. If elapsed before activation condition, runtime MUST invoke with **partial inputs** only if `allow_partial=true`; otherwise MUST route `INVALID_INPUT` to `on_error` policies.
- `allow_partial`: boolean (default false). If true, missing fields are filled using `defaults` (below) or schema defaults.
- `required_fields`: list of fully-qualified input paths required for `all_required` (default: all required fields in input schema).
- `defaults`: optional mapping of input field path -> default value.
- `conflict_resolution`: `reject` | `priority` | `deterministic` (default `reject`).
- `reject`: if two inbound edges attempt to set the same input field, runtime MUST reject the invoke attempt and route `INVALID_INPUT`.
- `priority`: runtime resolves conflicts using explicit per-edge priorities (see edge field `priority`).
- `deterministic`: runtime resolves conflicts deterministically (e.g., lowest `edge_id` lexical order). MUST be documented by the runtime.

Runtimes MUST implement at least `all_required` and `any`.

Runtimes MAY implement `quorum` in v0.2.x; if supported, it MUST be deterministic.

7.0.1 Carry State Acquisition Timing

The runtime MUST ensure that a Brick is invoked **only after** both:

1. the node activation condition is satisfied (Section 7.0), and
2. the node carry state is available according to the Brick's declared transport (Section 8.2).

Rules:

- If `carry_state_transport=inline`, the runtime MUST assemble the invocation envelope (including carry state) **after** the activation condition is met.
- If `carry_state_transport=handle`, the runtime MAY prefetch and prepare a `StateHandle` and a `carry_state_working_set` **before** the activation condition is met to reduce latency.
- The runtime MUST NOT hold **exclusive** leases on handle-based carry state until the activation condition is confirmed.
- The runtime MUST NOT invoke the Brick until the `StateHandle` is valid and any required working set is prepared.
- If carry state acquisition fails (e.g., lease contention, missing/expired state, storage unavailability), the runtime MUST treat the invocation as a `Failure` with `error_class=RESOURCE_EXCEEDED` and route according to `on_error` policies for that class.

This section prevents stale/missing carry-state invocations and avoids deadlocks caused by premature exclusive leasing.

7.1 Identity

Graph Manifest MUST include:

- `graph_id`
- `graph_version`
- `synaptic_state_version` (or equivalent version pin)

7.2 Nodes

Each node MUST define:

- `node_id`
- `brick`: {`brick_id`, `version_or_range`}
- `bindings.graph_ref_slot_values`: mapping of slot -> concrete ref source

Each node MAY define:

- `activation`: activation policy (see Section 7.0)
- `carry_state_lifecycle`: lifecycle policy (see Section 10.5)

7.3 Edges

Each edge MUST define:

- `edge_id` (stable within the graph manifest)
- `source_node`
- `target_node`
- `mapping`: typed field mapping from source output -> target input

Each edge MAY define:

- `priority`: integer (used only if node `conflict_resolution` is `priority`)

7.4 Routing Policies

Graphs MUST define routing for success and failures.

Note: Node activation policies (Section 7.0) define how inputs are assembled for fan-in before routing/invocation.

An edge MAY define:

- `on_success`: routing weight/threshold rules
- `on_error`: mapping from `error_class` to next node(s)

Graph runtime MUST implement **typed error routing**.

7.5 Budgets and Limits

Graph Manifest SHOULD define:

- max nodes per tick
 - max total ms per tick
 - max fan-out
 - max trace bytes
-

8. Invocation Envelope

8.1 Context (ctx)

Runtime MUST provide:

- trace_id
- session_id
- step
- graph_id, graph_version
- ctx.seed if determinism is stochastic

Trigger provenance (PIC-aligned): Runtime MUST include a trigger object describing the causal edge that triggered this invoke.

- trigger.source_node_id
- trigger.source_step
- trigger.edge_id
- trigger.routing_reason: on_success or on_error:<ERROR_CLASS>

If the invoke is the root of execution (external input), runtime MUST set trigger to a well-known root sentinel:

- trigger.source_node_id = "__root__"
- trigger.edge_id = "__root__"

Time injection depends on time_model:

- logical_time for logical
- wall_time for wall_bounded

8.2 Carry State Delivery

- If transport is inline: runtime passes carry_state value.
- If transport is handle: runtime passes:
 - carry_state_handle
 - optional carry_state_working_set

8.3 Synaptic State Delivery

Runtime provides `graph_refs` based on slot bindings:

- `graph_refs`: mapping slot -> value
-

9. Result Model and Typed Errors

9.1 Result Union

Brick invocation MUST return exactly one of:

- `Success { output, carry_state_next, obs_events, metadata }`
- `Failure { error, obs_events, metadata, carry_state_side_effects? }`

9.2 Structural Boundary Rule (Mandatory)

- If `output` is **present**, `error_class` MUST be `LOW_CONFIDENCE`.
- If `output` is **absent**, `error_class` MUST NOT be `LOW_CONFIDENCE`.

Runtime MUST validate and reject nonconforming results.

9.3 Error Object

`error` MUST include:

- `error_class` (enum)
- `message` (short, redaction-safe)
- `retry_advice`: `NEVER | IMMEDIATE | BACKOFF`
- `severity`: `WARN | ERROR | FATAL`

9.4 Error Classes (v0.2)

- `INVALID_INPUT`: semantic invalid; no output produced
- `COMPUTATION_ERROR`: trap/exception/numerical failure; no output produced
- `RESOURCE_EXCEEDED`: hit declared limits; no output produced
- `RUNTIME_REJECTED`: policy/manifest violation; no output produced
- `LOW_CONFIDENCE`: output produced + flagged below confidence threshold

9.5 Confidence Signaling

If `error_class=LOW_CONFIDENCE`, output MUST include:

- `confidence` scalar in [0,1] (or a declared alternative)

Brick Manifest SHOULD declare:

- `confidence_threshold` (default used by graph policies)
-

10. Carry State Storage Semantics (Runtime-owned)

10.1 CAS and ETags

Carry state updates MUST use compare-and-swap (CAS):

- runtime provides `etag`
- Brick proposes `expected_etag`
- runtime commits only if match; else returns conflict

10.2 Conflict Behavior

Default behavior MUST be **conflict-reject**. Graphs requiring merges MUST model merges explicitly via merge Bricks at fan-in points.

10.3 Leases (Optional but Recommended)

Runtime MAY implement leases to reduce contention:

- `AcquireLease(session_id, owner, lease_ms)`
- leased owners may batch checkpoints

10.4 StateHandle

`StateHandle` MUST be opaque to the Brick. Runtime MUST ensure that handle-based state does not violate carry-state size constraints and TTL policies.

10.5 Session and Carry State Lifecycle (Normative)

The runtime MUST define when session-scoped carry state is created, initialized, expired, and how it behaves across graph version changes.

A Graph Manifest MAY define per-node `carry_state_lifecycle`:

- `init`: `zero` | `from_graph_defaults` | `from_first_input`
- `zero`: runtime initializes carry state to the zero/empty value implied by the `carry_state` schema.
- `from_graph_defaults`: runtime initializes carry state from defaults embedded in the graph manifest.
- `from_first_input`: runtime initializes carry state on first invoke using a deterministic function of the first input (MUST be documented by runtime or implemented as an explicit init Brick).

- `ttl_seconds`: integer. Runtime MUST expire carry state for the node after this many seconds since last access.
- `on_graph_version_change`: `reset` | `migrate_if_compatible` | `reject_session`
- `reset` (default): runtime MUST reset carry state when `graph_version` changes.
- `migrate_if_compatible`: runtime MAY carry over state only if the Brick version and `carry_state` schema are unchanged and declared patch-compatible.
- `reject_session`: runtime MUST continue serving the session on the pinned `graph_version` or fail-closed if that version is unavailable.

Defaults if `carry_state_lifecycle` is omitted:

- `init = zero`
- `ttl_seconds = implementation_default` (runtime MUST document)
- `on_graph_version_change = reset`

Runtime MUST treat carry state as scoped to `(session_id, node_id, graph_version)` unless `migrate_if_compatible` is in effect.

11. Observability and Tracing

11.1 Minimal Trace Record (Mandatory)

Runtime MUST record at least:

- `timestamp` (runtime time)
- `trigger.source_node_id`, `trigger.source_step`, `trigger.edge_id`, `trigger.routing_reason`
- `trace_id`, `session_id`, `step`
- `graph_version`
- `brick_id`, `brick_version`, `bundle_digest`
- `input_hash`, `carry_state_hash` (if present), `output_hash` (if present)
- `error_class` (if Failure)
- `latency_ms`

Additional requirements:

- If `time_model=wall_bounded`, runtime MUST include `ctx.wall_time` (or its hash) in trace.
- If a Brick declares `determinism.model_pin` (Section 6.6.1), runtime MUST include:
 - `determinism.model_pin`
 - `determinism.reproducibility_level`
- If `determinism.reproducibility_level=seed_approximate`, runtime MUST additionally record either:
 - the **full output payload** (e.g., CBOR-encoded output), OR
 - a stable `output_ref` pointing to an operator-controlled secure audit store containing the full output.

11.2 obs_events

Bricks MAY emit `obs_events` as append-only records. Runtime MUST treat `obs_events` as write-only from Brick perspective. Runtime SHOULD cap per-invoke `obs_events` bytes.

12. Cache and Replay Semantics

12.1 Cache Key

For deterministic modes, runtime MAY cache results. Cache key MUST include:

- `bundle_digest`
- `graph_version`
- `input_hash`
- `carry_state_hash` (if used)
- `graph_refs_hash` (values of slots used)
- `ctx.logical_time` if `time_model=logical`
- `ctx.wall_time` if `time_model=wall_bounded` (or recorded wall input)

If the Brick declares `determinism.model_pin` (Section 6.6.1):

- Cache key MUST include `determinism.model_pin`.
- If `determinism.reproducibility_level=seed_approximate`, runtime MUST NOT serve cached results for that Brick.

12.2 Replay

A replay engine MUST be able to reproduce decisions by re-feeding recorded inputs + pinned versions + recorded time inputs.

13. Registry and Lifecycle

13.1 Publish and Trust

A registry SHOULD support:

- publish bundle
- fetch by `brick_id@version`
- verify signature and attestations

Runtimes SHOULD support policy:

- allowlist issuers
- require determinism conformance attestations for certain determinism modes

13.2 Version Policy

Manifest MAY declare `version_policy`:

- `patch_compatible`: patch updates MUST preserve schemas and compatibility

Graphs MAY reference:

- exact versions (recommended for production)
- version ranges (allowed in dev; runtime resolves to pinned version at serve time)

13.3 Deprecation

If `status=sunset`, runtime SHOULD warn or reject serving depending on operator policy. If `successor` is provided, tooling SHOULD suggest migration.

14. Security Considerations

14.1 Sandbox

Runtime MUST enforce no ambient authority and resource limits.

14.2 Capability Model

- graph ref access is only via declared slots
- time inputs only via declared `time_model`

14.3 Information Boundaries

Bricks MUST NOT be able to introspect graph topology beyond declared ref slots.

14.4 Supply Chain

Operators SHOULD require signature verification and may require attestations.

15. Conformance Suites (Recommended v0.2)

15.1 Determinism Conformance

- `bit_exact`: byte-identical outputs on reference runner
- `bounded_drift`: within epsilon across reference runs
- `stochastic`: deterministic given identical seed

15.2 Sandbox Profile Conformance

- verify forbidden syscalls/capabilities
- verify time/rng restrictions

15.3 Schema Stability

- verify `patch_compatible` updates preserve schemas

15.4 Wire Compatibility

- verify canonical encoding stability for envelope/result
- verify WASM ABI compliance

16. Wire Format and Transport (Normative)

16.1 Normative Encoding

NCP defines a **normative binary wire format** for the invocation envelope and result:

- **CBOR** (RFC 8949) MUST be supported.
- **Deterministic (canonical) CBOR encoding** per RFC 8742 MUST be used for any bytes that are hashed/digested or signed.

JSON MAY be supported for tooling/debugging but MUST NOT be used for digest/signature computation.

16.2 WASM ABI (v0.2 REQUIRED)

To ensure Brick portability, all WASM Bricks MUST implement the same entrypoint ABI.

Entrypoint function name: `invoke` (or as specified by manifest `artifact.entrypoint`).

Function signature (32-bit pointers):

- `invoke(envelope_ptr: i32, envelope_len: i32) -> result_ptr: i32`

Where:

- The invocation envelope is a CBOR buffer located in linear memory at `[envelope_ptr, envelope_ptr + envelope_len)`.
- The result is a CBOR buffer whose pointer is returned as `result_ptr`.

Result buffer format:

- The first 4 bytes at `result_ptr` MUST be `result_len` as a little-endian `u32`.
- The CBOR result bytes MUST begin at `result_ptr + 4` and be `result_len` bytes.

16.3 Memory Management

Runtimes MUST provide imports for allocation/free to avoid leaking memory across invocations.

One of the following memory models MUST be implemented by a compliant runtime:

- **Model A (Runtime allocates, Brick writes):** runtime provides an output buffer pointer/len via an import; Brick writes into it.
- **Model B (Brick allocates, Runtime frees):** Brick uses exported `alloc(len)`; runtime calls exported `free(ptr, len)` after reading.

v0.2 RECOMMENDS Model B:

- Brick MUST export `alloc(len: i32) -> i32` and `free(ptr: i32, len: i32) -> void`.
- Runtime MUST call `free` on the result buffer after consumption.

16.4 Transport

NCP does not mandate a network transport in v0.2. Implementations MAY use:

- in-process calls
- shared memory
- gRPC/HTTP2
- QUIC

However, the **payload encoding and WASM ABI** are normative and MUST remain identical across transports.

Appendix C. Integration Patterns (Non-normative)

C.1 Exposing an NCP Graph as an MCP Tool

An NCP Graph MAY be exposed as a Model Context Protocol (MCP) tool by mapping:

- the Graph entry node's input schema → the MCP tool `inputSchema`
- the Graph exit node's output schema → the tool output schema (if used by the MCP client)
- the Graph identity/metadata (`graph_id`, description) → the MCP tool name/description

From the MCP client's perspective, the Graph appears as a single tool call. Internally, the NCP runtime executes the full Brick network (routing, fan-in/fan-out, and typed error recovery remain internal to NCP).

C.2 Using MCP Servers as Brick Backends (Future Extension)

A Brick MAY delegate computation to an external MCP server only in a future NCP version that extends `capabilities[]` beyond empty to include an explicit, enforceable network capability (e.g., `mcp_client`).

Such Bricks would be expected to:

- declare `determinism.mode: stochastic` and the appropriate `reproducibility_level` (Section 6.6.1)
- obey strict runtime-enforced allowlists, timeouts, and resource limits

v0.2 does not permit networked Brick backends because `capabilities[]` MUST be empty (Section 6.5).

Appendix A. Example Brick Manifest (YAML)

```
brick_id: org.ncp-commons.sentiment-gate
version: 1.4.2
ncp_protocol_range: ">=0.2.0 <0.3.0"
required_runtime_features: ["logical_clock"]
status: active

artifact:
  format: wasm
  entrypoint: invoke
  digest: "sha256:..."
  size_bytes: 47832

schemas:
  input: schemas/input.json
  output: schemas/output.json
  carry_state: null

limits:
  max_ms: 2
  max_mem_mb: 32
  max_output_bytes: 4096
  max_input_bytes: 4096
  estimated_cost_per_invoke_usd: 0.000003

capabilities: []

determinism:
  mode: bounded_drift
  epsilon: 1.0e-5

time_model: logical

carry_state_class: none
carry_state_transport: inline
carry_state_max_bytes: 0
```

```

carry_state_side_effects_schema: null

confidence_threshold: 0.70

graph_ref_slots:
  - slot: SENTIMENT_ROUTE_PRIOR
    type: float
    access: read
    ref_consistency: PINNED

```

Appendix A.2 Example External-Model Stochastic Brick (YAML)

```

brick_id: org.acme.llm-escalation
version: 0.3.0
ncp_protocol_range: ">=0.2.0 <0.3.0"
required_runtime_features: ["wall_clock"]
status: active

artifact:
  format: wasm
  entrypoint: invoke
  digest: "sha256:..."
  size_bytes: 152004

schemas:
  input: schemas/input.json
  output: schemas/output.json
  carry_state: null

limits:
  max_ms: 3000
  max_mem_mb: 256
  max_output_bytes: 16384
  max_input_bytes: 32768
  estimated_cost_per_invoke_usd: 0.003

capabilities: []

determinism:
  mode: stochastic
  seed_source: ctx.seed
  model_pin: "provider:example-llm;model:sonnet-like;version:2025-05-14"
  reproducibility_level: seed_approximate

time_model: wall_bounded

```

```

carry_state_class: none
carry_state_transport: inline
carry_state_max_bytes: 0

carry_state_side_effects_schema: null

# This Brick typically emits structured output such as:
# - escalation_decision
# - draft_reply
# - rationale (optional, policy-controlled)

graph_ref_slots: []

```

Note (informational): For `seed_approximate` Bricks, runtimes MUST NOT serve cached results and MUST record the full output payload (or an `output_ref` to a secure audit store) per Sections 6.6.1 and 11.1.

Appendix A.2.1 Example Input Schema (JSON Schema, non-normative)

```

{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "LlmEscalationInput",
  "type": "object",
  "additionalProperties": false,
  "required": ["query", "policy"],
  "properties": {
    "query": {
      "type": "string",
      "description": "End-user request or message to escalate."
    },
    "context": {
      "type": "object",
      "description": "Optional structured context produced by upstream Bricks.",
      "additionalProperties": true,
      "properties": {
        "sentiment": {"type": "number", "minimum": -1, "maximum": 1},
        "urgency": {"type": "number", "minimum": 0, "maximum": 1},
        "customer_tier": {"type": "string"},
        "signals": {
          "type": "array",
          "items": {"type": "string"}
        }
      }
    },
    "policy": {

```

```

    "type": "object",
    "additionalProperties": false,
    "required": ["mode"],
    "properties": {
      "mode": {
        "type": "string",
        "enum": ["draft_reply", "classify_only", "route_only"],
        "description": "Controls what the Brick is allowed to output."
      },
      "max_output_tokens": {"type": "integer", "minimum": 1},
      "allow_rationale": {"type": "boolean", "default": false}
    }
  }
}

```

Appendix A.2.2 Example Output Schema (JSON Schema, non-normative)

```

{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "LlmEscalationOutput",
  "type": "object",
  "additionalProperties": false,
  "required": ["decision"],
  "properties": {
    "decision": {
      "type": "string",
      "enum": ["respond", "escalate_human", "request_more_info"],
      "description": "High-level decision produced by the LLM Brick."
    },
    "draft_reply": {
      "type": "string",
      "description": "Optional draft reply for downstream review or user response."
    },
    "tags": {
      "type": "array",
      "items": {"type": "string"},
      "description": "Optional routing tags (e.g., billing, refund, bug)."
    },
    "rationale": {
      "type": "string",
      "description": "Optional explanation. Typically gated by policy.allow_rationale."
    }
  }
}

```

```
}  
}
```

Appendix B. Example Graph Slot Binding (YAML)

```
graph_id: org.acme.support-routing  
graph_version: "sha256:..."  
synaptic_state_version: "sha256:..."  
  
nodes:  
  - node_id: sentiment_gate  
    brick: { brick_id: org.ncp-commons.sentiment-gate, version_or_range:  
      "=1.4.2" }  
    activation:  
      mode: all_required  
      timeout_ms: 50  
      allow_partial: false  
      conflict_resolution: reject  
    bindings:  
      graph_ref_slot_values:  
        SENTIMENT_ROUTE_PRIOR: { ref: "edge_weight(sentiment_gate->router)",  
consistency: PINNED }  
  
  - node_id: router  
    brick: { brick_id: org.ncp-commons.multi-signal-router, version_or_range:  
      "=2.1.0" }  
    activation:  
      mode: all_required  
      timeout_ms: 50  
      allow_partial: true  
      required_fields: ["input.sentiment", "input.urgency",  
      "input.customer_tier"]  
    defaults:  
      input.customer_tier: "free"  
      conflict_resolution: reject  
  
edges:  
  - edge_id: sentiment_gate_to_router  
    source_node: sentiment_gate  
    target_node: router  
    mapping:  
      - from: output.label  
        to: input.sentiment  
    on_success: { weight: 0.92, threshold: 0.70 }  
    on_error:  
      INVALID_INPUT: [fallback_classifier]
```

```
LOW_CONFIDENCE: [llm_escalation]  
COMPUTATION_ERROR: [redundant_sentiment_gate]  
RESOURCE_EXCEEDED: [queue_for_review]  
RUNTIME_REJECTED: [fail_closed]
```