BRIDGE OS

Technical Architecture Brief

For Engineering Review

1. OVERVIEW

Bridge OS is a governance middleware that enforces human authority over AI-assisted decisions. It implements structural incompletion: the system cannot complete certain actions without human intervention. This is not a behavioral suggestion. It is architectural enforcement at the schema level.

The core principle is that any system completing meaning autonomously destroys the meaning it claims to complete. A remainder must be preserved for human judgment. Bridge OS enforces this remainder through gates that cannot be bypassed by code.

2. THE CORE FUNCTION

All governance decisions flow through a single deterministic function. No AI judgment. No probabilistic behavior. Just rules.

Function Signature:

```
export function evaluate(
  session: Session,
  artifacts: Artifact[],
  action: Action,
  actor: Actor
): EvalResult {
  const errors: EvalError[] = [];

  // 1) Schema validation
  errors.push(...validateSession(session));
  errors.push(...validateArtifacts(artifacts));

  // 2) Human-only gates
  errors.push(...humanOnlyGate(actor, action));

  // 3) Transition policy
  errors.push(...transitionPolicy(session, artifacts, action));

  // If any errors, block the action
  if (errors.length > 0) {
    return { allowed: false, errors, newSession: session, newArtifacts: artifacts };
  }

  // Otherwise, apply the action and return new state
  // ...
}
```

The function returns one of two outcomes: allowed (with new state) or blocked (with error codes). Same inputs always produce same outputs. This is deterministic governance.

3. THE STATE MACHINE

Sessions progress through defined states. Each transition has requirements that must be met.

Session States:

```
DRAFT: Initial state. No compilation has begun.

IN_COMPILATION: Active work. Artifacts are being created and approved.

SEEDSWEEP_IN_PROGRESS: Pause state. A reset/review cycle is active.

AWAITING_HUMAN_SIGNOFF: All artifacts approved. Waiting for human to sign off.

FINALIZED: Complete. No further changes allowed.
```

Transition Flow:

```
DRAFT --> IN_COMPILATION
  Requires: SeedSweep artifact APPROVED and GREEN

IN_COMPILATION --> AWAITING_HUMAN_SIGNOFF
  Requires: All core artifacts (Ingestion, Semantic, Execution, Governance) APPROVED
  Requires: No RED stoplights

AWAITING_HUMAN_SIGNOFF --> FINALIZED
  Requires: human_signoff.signed = true
  Requires: Actor = HUMAN
```

4. THE GATES

Three types of gates enforce governance. Each gate is checked by the evaluate function before any action proceeds.

Human-Only Gate:

Certain actions can only be performed by a HUMAN actor. If an ASSISTANT attempts these actions, the system returns HUMAN_GATE_REQUIRED.

```
// Human-only actions:
// - ARTIFACT_SET_STATUS to APPROVED or REJECTED
// - SESSION_SIGNOFF
// - SESSION_FINALIZE

export function humanOnlyGate(actor: Actor, action: Action): EvalError[] {
  const isApproval = action.type === "ARTIFACT_SET_STATUS"
    && (action.status === "APPROVED" || action.status === "REJECTED");
  const isSignoff = action.type === "SESSION_SIGNOFF";
  const isFinalize = action.type === "SESSION_FINALIZE";

  if (actor === "ASSISTANT" && (isApproval || isSignoff || isFinalize)) {
    return [{ code: "HUMAN_GATE_REQUIRED", message: "This action requires human authority." }];
  }
  return [];
}
```

SeedSweep Preflight Gate:

Before compilation can begin, a SeedSweep artifact must be APPROVED and have a GREEN stoplight. This ensures a human has reviewed the session before work proceeds.

```
// SeedSweep preflight check
```

```
if (action.type === "SESSION_START_COMPILATION") {
  const seedSweep = artifacts.find(a => a.compiler === "SEEDSWEEP");

  if (!seedSweep || seedSweep.status !== "APPROVED") {
    errors.push({ code: "SEEDSWEEP_PREFLIGHT_REQUIRED", message: "..." });
  }

  if (seedSweep && seedSweep.stoplight !== "GREEN") {
    errors.push({ code: "SEEDSWEEP_NOT_GREEN", message: "..." });
  }
}
```

Stoplight Binding:

The session stoplight is bound to the worst stoplight among its artifacts. If any artifact is RED, the session cannot proceed to finalization.

```
// Stoplight hierarchy: RED > YELLOW > GREEN
export function worstStoplight(stoplights: Stoplight[]): Stoplight {
  if (stoplights.includes("RED")) return "RED";
  if (stoplights.includes("YELLOW")) return "YELLOW";
  return "GREEN";
}

// Applied during evaluation:
const coreArtifacts = artifacts.filter(a =>
  ["INGESTION", "SEMANTIC", "EXECUTION", "GOVERNANCE"].includes(a.compiler)
);
const sessionStoplight = worstStoplight(coreArtifacts.map(a => a.stoplight));
```

5. THE SCHEMAS

Data structures are validated using Zod schemas. Invalid data is rejected before gate logic runs.

Session Schema:

```
export const SessionSchema = z.object({
  id: z.string().min(1),
  createdAt: z.string().min(1),
  updatedAt: z.string().min(1),
  state: z.enum(["DRAFT", "IN_COMPILATION", "SEEDSWEEP_IN_PROGRESS",
                 "AWAITING_HUMAN_SIGNOFF", "FINALIZED"]),
  stoplight: z.enum(["GREEN", "YELLOW", "RED"]),
  title: z.string().min(1),
  intent: z.string().min(1),
  holding_space_003: z.array(z.string().min(1)).min(1),  // Required, non-empty
  human_signoff: HumanSignoffSchema,
  activeSeedSweepArtifactId: z.string().min(1).optional()
});
```

Artifact Schema:

```
export const ArtifactSchema = z.object({
  id: z.string().min(1),
  compiler: z.enum(["INGESTION", "SEMANTIC", "EXECUTION", "GOVERNANCE", "SEEDSWEEP"]),
  status: z.enum(["DRAFT", "IN_REVIEW", "APPROVED", "REJECTED"]),
  stoplight: z.enum(["GREEN", "YELLOW", "RED"]),
  payload: z.union([...]),  // Compiler-specific payload
  holding_space_003: z.array(z.string().min(1)).min(1),  // Required, non-empty
  approvals: z.object({
    approvedByHuman: z.boolean(),
    approvedAt: z.string().min(1).optional(),
    approvedBy: z.string().min(1).optional()
  })
});
```

The holding_space_003 field is required on both Session and Artifact. It must contain at least one non-empty string. This enforces structural incompletion: every object must acknowledge what remains open or unresolved.

6. WHY IT IS STRUCTURAL

The distinction matters. Behavioral compliance means the system chooses to follow rules. Structural enforcement means the system cannot proceed without meeting requirements.

When an ASSISTANT attempts to approve an artifact, the humanOnlyGate function returns an error. The evaluate function sees errors and returns allowed: false. The state does not change. There is no path through the code that allows an ASSISTANT to approve an artifact.

This is not about trusting the AI to behave. This is about the schema rejecting invalid state. A wolf cannot convince the schema. A wolf cannot find an alternative path. A wolf hits the wall and cannot proceed.

7. HOW TO TEST

A self-verifying harness runs 16 governance steps. Each step either passes or the harness fails immediately.

Running the Demo:

```
# Navigate to the project folder
cd bridge-os-governance

# Install dependencies (first time only)
npm install

# Run the demo harness
npx tsx demo/run.ts
```

What the Harness Tests:

```
Step 1: Attempt compilation before SeedSweep approved (should BLOCK)
Step 2: Assistant attempts to approve SeedSweep (should BLOCK)
Step 3: Human approves SeedSweep (should ALLOW)
Step 4: Attempt compilation with SeedSweep YELLOW (should BLOCK)
Step 5: Human sets SeedSweep GREEN (should ALLOW)
Step 6: Start compilation (should ALLOW)
Step 7: Assistant attempts to approve Ingestion (should BLOCK)
Steps 8-11: Human approves all core artifacts (should ALLOW)
Step 12: Request finalize (should ALLOW)
Step 13: Assistant attempts signoff (should BLOCK)
Step 14: Human signoff (should ALLOW)
Step 15: Assistant attempts finalize (should BLOCK)
Step 16: Human finalize (should ALLOW)
```

Output Files:

The harness generates two files in demo/output/: transcript.jsonl: Machine-readable audit log. Every step, every action, every result. timeline.md: Human-readable summary of the test run.

Adversarial Testing:

Try to break it. Modify the fixtures. Change the actor. Add new actions. If you find a path that bypasses a human gate, that is a bug. If you cannot find such a path, that is the proof.