Tall-Infra, Data-Only Development

A Technical Paper for Game Developers

Registry: [HOWL-COMP-2-2026]

Series Path: [HOWL-COMP-1-2026] → [HOWL-COMP-2-2026] → [HOWL-COMP-3-2026] → [HOWL-COMP-4-2026] → [HOWL-COMP-5-2026]

Parent Framework: [HOWL-COMP-1-2026]

DOI: 10.5281/zenodo.zzz

Date: February 2026

Domain: Software Architecture / Systems Engineering / Real-Time Computing

Status: Architectural Blueprint for Independent Implementation

AI Usage Disclosure: Only the top metadata, figures, refs and final copyright sections were edited by the author. All paper content was LLM-generated using Anthropic's Claude 4.5 Sonnet.

Abstract

This paper defines and distinguishes **tall-infra, data-only** architecture from the commonly used term "data-driven." While data-driven development has been industry standard for decades, it has always preserved a wall between data (assets, tuning values) and behavior (compiled code). Tall-infra, data-only eliminates this wall entirely. The runtime binary contains no gameplay types, no behavior code, no knowledge of what a "goblin" or "quest" or "fireball" is. All semantics—every enemy, mechanic, system interaction—exist purely as dataset rows that can be hot-swapped without recompilation.

This is not a proposal. It describes a working system running 200 entities with 100 followers each at 60fps, unoptimized, with a clear path to 10,000+ entities.

---

1. The Core Distinction

Data-driven means: code owns the types, data owns the numbers.

Data-only means: the binary forgets what a "goblin" is; the dataset teaches it every frame.

Tall-infra, data-only means: the previous sentence is true from renderer to AI to UI to audio—no exceptions, across the entire vertical stack.

---

2. Historical Context

Every major engine era has claimed "data-driven" while preserving compiled behavior somewhere.

1990s: Quake, Half-Life

- **What was data**: sounds, models, textures
- **What was still code**: AI routines, weapon behavior, level triggers
- **Wall present**: Yes

2000s: Unreal, Unity

- **What was data**: meshes, prefabs, material parameters
- **What was still code**: gameplay classes, damage calculations, quest logic
- **Wall present**: Yes

2010s: Stingray, Overwatch ECS

- **What was data**: entities, component values
- **What was still code**: AI behaviors, hero-specific abilities, system logic
- **Wall present**: Yes

2020s: Bevy, Unity DOTS

- **What was data**: component columns, archetype definitions
- **What was still code**: systems that process components
- **Wall present**: Yes

In every case, somewhere in the binary lives compiled behavior. Boss attack patterns. Damage formulae. Pathfinding heuristics. Dialogue state machines. The wall moved over the decades, but it never disappeared.

---

3. Definition: Tall-Infra, Data-Only

A project qualifies as tall-infra, data-only when **all** of the following are true:

3.1 Zero Gameplay Types in the Binary

No `class Goblin`. No `struct QuestReward`. No `enum FireSpell`. The compiled code contains infrastructure only: state machine runners, predicate evaluators, utility scorers, effect processors. It does not know what game it is running.

3.2 Behavior Lives in Hot-Swappable Data Tables

State machines, AI decision curves, logic rules, stat modifications, event mappings—all stored as table rows. Change a row, change the behavior. No recompilation. No restart required.

3.3 Designers Can Invent New Mechanics Without Engineers

Not "tweak values" but "create new systems." A designer can use the equipment-slot table to store camera FOV for a security terminal. They can repurpose the health stat as a permission bitmap. Cross-wiring existing tables creates emergent mechanics.

3.4 Every Subsystem Consumes the Same Uniform Data Layout

Rendering, physics, AI, audio, UI, networking, save/load—all read from the same column-oriented tables with foreign key relationships. No special cases. No subsystem-specific formats.

---

4. The Architecture Stack

A tall-infra, data-only system requires specific infrastructure layers.

4.1 Entity: The Universal Container

An entity is not a type. It is a uniform struct containing optional capability systems:

```
Entity:
  id: i32
  from_entity_id: i32  // template reference
  state_machine_id: i32
  state: Text


 // Optional systems - all have defaults, all can be repurposed
  visual: VisualSystem
  movement: MovementSystem
  combat: CombatCapabilities
```

```
container: ContainerSystem

behavior: AIBehavior

// ... 20+ more systems
```

A dragon, a chair, a weather controller, and a bank account are all the same struct. The difference is which `state_machine_id` they reference and which table rows populate their systems.

4.2 State Machine: Pure Topology

States do not contain behavior code. They contain:

- A name
- A reference to a behavior set (for AI scoring)
- Transitions with exit conditions (referencing predicate rules)
- Optional forced actions (skill activation)

One runner function walks the state graph. Whether it's evaluating "dragon combat phases" or "shop open/closed hours" or "weather patterns" is determined entirely by data.

4.3 Predicate Logic: Industrial-Strength Conditions

Traditional if/else/switch statements are singular and stateful. They don't scale to complex multi-entity queries.

Predicate logic (Prolog-style rules) provides:

```
Rule: can_attack(X, Y)

Body:

  - is_hostile(X, Y)

  - in_range(X, Y, 50)

  - has_line_of_sight(X, Y)

  - not(is_stunned(X))
```

Rules are evaluated against a fact set generated each frame. The evaluator doesn't know if it's checking combat eligibility or trade permissions or quest prerequisites—it unifies predicates against facts.

4.4 Utility AI: Behavior Scoring

Behaviors are scored, not scripted. Each behavior has considerations:

```
Behavior: "retreat"

Considerations:

  - health: range [0,100], curve: inverse_linear, weight: 1.5

  - enemy_count: range [0,5], curve: exponential, weight: 1.0

  - distance_to_safety: range [0,100], curve: linear, weight: 0.8
```

The utility scorer multiplies normalized, curved values. Highest score wins. What "retreat" means—which action fires, which logic executes—is determined by data references, not compiled code.

4.5 Logic Blocks: Stack-Based Effects

When mutation is needed, logic blocks provide a bytecode interpreter:

```
BlockStack: "apply_fire_damage"

Rows:

   - DrGetFloat: "target.stats.fire_resistance"

   - MathSub: 1.0, [previous]

   - MathMultiply: [previous], base_damage

  - DrSetFloat: "target.stats.health", [subtract previous]
```

The runner executes arithmetic and data operations. It doesn't know if it's applying fire damage or adjusting bank interest or calculating weather intensity.

4.6 Envelopes: Automated Stat Transformation

Effects that modify stats over time use envelopes:

```
ActionSkillStatEnvelope:

   source_entity_id: 47

   target_entity_id: 23
```

```
stat: 9000  // just an i32

modifier: -50

curve: Linear

time_start: 10.5

time_end: 15.5
```

The envelope processor applies modifiers each frame based on elapsed time and curve shape. Whether stat 9000 is "health" or "reputation" or "temperature" is a dataset concern.

---

5. Concrete Example: Dragon Boss in Four Rows

Traditional approach requires:

- `class DragonBoss : Enemy`
- Phase transition logic in code
- Attack pattern methods
- Health bar UI binding
- Sound effect triggers
- Particle system spawning

Tall-infra, data-only approach:

Table	Key Fields	Runtime Sees
game.entity	name="Dragon", state_machine_id=97	generic entity
game.state_machine	id=97, states=[phase1, phase2, phase3]	any state machine
game.behavior_set	id=42, behaviors=[breath_attack, tail_swipe, summon_adds]	any behavior set
game.stat	entity_id=dragon, stat_type=9000, value=5000	any stat column

The binary processes a state machine, scores behaviors, applies envelopes. It never executes "dragon code" because no such code exists.

---

6. Concrete Example: Equipment Slot as Database Column

The container system has equipment slots:

```
EquipmentSlot:

  slot_type: EquipmentSlotType  // i32 enum

  world_item_id: i32

  count: i32

  health_max: i32

  health: i32
```

Traditional use: `slot_type = .HandRight`, `world_item_id = sword_47`

Data-only repurposing:

```
slot_type = .Backpack

world_item_id = 334  // points to row "player_bank_balance"

count = 50000        // balance amount

health_max = 10000   // insurance coverage cap

health = 10000       // current coverage
```

The bank UI sums `count * world_item.value`. The `health` field becomes insurance that decays on withdrawal. No code change. Same infrastructure, different semantics.

---

7. Concrete Example: Weather, Morale, and Crafting

Three apparently unrelated systems:

- Weather state progression (sunny → cloudy → rainy → stormy)
- NPC morale states (confident → nervous → panicked → broken)
- Crafting tech tree (locked → researching → unlocked → mastered)

All three use the same state machine runner. All three use utility scoring to pick transitions. The weather system's "current behavior" isn't executed—it's read as the current

weather state. The morale system's behavior triggers AI actions. The crafting system's state gates recipe availability.

One function evaluates transitions. One function scores behaviors. Semantics differ entirely based on how the data is wired.

---

8. Anti-Examples: What This Is Not

8.1 Unity Scriptable Objects

Claim: "Our data is in ScriptableObjects, we're data-driven!"

Reality: You still write `class Goblin : MonoBehaviour` with `void Attack()` methods. The ScriptableObject holds stats; the behavior is compiled.

Wall present: Yes. New enemy types require new classes.

8.2 Unreal Data Tables

Claim: "We use Data Tables for everything!"

Reality: Damage calculations are Blueprint or C++. Quest branches are Blueprint or C++. AI behavior trees have compiled task nodes.

Wall present: Yes. New mechanics require new nodes.

8.3 Overwatch-Style ECS

Claim: "Pure ECS, all data in components!"

Reality: Hero-specific systems contain compiled logic. Reinhardt's charge, Tracer's blink, Mercy's resurrect—all have dedicated code paths.

Wall present: Yes. New heroes require new systems.

8.4 Factorio Prototypes

Claim: "Everything is defined in prototype data!"

Reality: Combat AI is compiled. Train pathfinding is compiled. Electric network solving is compiled. Prototypes define *what exists*; code defines *how it behaves*.

Wall present: Yes. New transport types require new code.

---

9. The Litmus Test

Answer honestly:

1. Can a designer add a "living market economy" without asking engineering?

2. Can `stat.health` be repurposed as "bank account balance" with zero code change?
3. Can you swap every texture, sound, and behavioral rule while the executable keeps running?
4. Does the compiled binary know that any noun (goblin, sword, quest, fireball) exists?

If any answer is "no," you have a wall. You may be data-driven. You are not data-only.

---

10. Debugging and Observability

A common objection: "If everything is data, how do you debug? Stack traces become meaningless."

The solution is structured tracing at every decision point:

```
TraceType:

  // State machine layer

  state, state_transition, state_transition_rule


  // Utility AI layer

  behavior_set, behavior_score, behavior_selection

  consideration, consideration_rule


  // Logic block layer

  block_stack, block_stack_result, block_condition

  block_rule, block_execute_command


  // Mutation layer

  set_dr, set_entity_fact
```

Each entity maintains a trace of decisions per frame. "Why did the dragon breathe fire twice?" becomes a query:

```
Entity 47, frame 1842:
  state_transition: phase1 → phase2 (rule: health_below_50 passed)
  behavior_selection: breath_attack (score: 0.87)
  envelope_created: stat 9000, modifier -200, target entity 12
```

This is superior to stack traces. You're debugging game logic in game terms, not memory addresses.

---

11. Performance

Theoretical concern: "Runtime indirection must be slow."

Measured reality: 200 entities $\times$ 100 followers $\times$ 60fps = 20,000 transform calculations per frame, unoptimized.

The architecture:

- All data in RAM (memdb)
- O(1) column access
- Per-frame fact generation
- Per-frame entity pair calculations (distance, facing, line of sight)
- SQLite as persistence layer, not runtime layer
- Dirty/new rows serialized on demand

Prolog unification over small fact sets is cheap. Utility scoring is arithmetic. Envelope processing iterates a fixed array. The "indirection cost" is negligible compared to rendering and physics.

Optimization target: 10,000+ entities with 100 followers each.

---

12. Productivity Implications

Metric	Traditional	Tall-Infra Data-Only
New enemy type	Code + data + test	Data only
New mechanic	Design + code + iterate	Data wiring

Metric	Traditional	Tall-Infra Data-Only
Balance pass	Rebuild or hot-reload values	Hot-swap tables
Multi-project reuse	Fight the old architecture	Same binary, new dataset
Live service patch	Risk regression in compiled behavior	Zero code risk

The marginal cost of new content approaches zero engineering time. The 1000th behavior set costs the same as the 10th. Complexity remains constant because interactions are data relationships, not code dependencies.

---

13. Learning Curve

This architecture requires new sub-disciplines:

Data-relational thinking: Everything is foreign keys and columns. A "dragon" is a row that references other rows.

Predicate logic: Conditions are rules unified against facts, not if/else chains.

Utility curve design: Behaviors emerge from scored considerations, not scripted sequences.

Envelope-based effects: Modifications are time-bounded stat transformations, not imperative mutations.

Trace-driven debugging: Problems are queries over decision history, not breakpoints in code.

These are learnable. They are also better mental models than "write code for each thing." The designer who thinks in curves and predicates expresses more with less.

---

14. When to Use This

Strong fit:

- Live service games with continuous content updates
- Multi-SKU studios amortizing infrastructure across projects
- Games with heavy modding support
- Large content volume (hundreds of enemy types, thousands of items)
- Long development cycles where requirements will change

Weak fit:

- Game jams (no time to build infrastructure)
- Prototypes (exploration over architecture)
- Single-ship titles with small teams and fixed scope
- Games where performance requires hand-tuned compiled code paths

The infrastructure cost is front-loaded. The productivity multiplier is permanent.

---

15. Conclusion

"Data-driven" has meant "faster tuning" for thirty years. The wall between data and behavior remained.

"Data-only" means faster content creation. Behavior itself becomes data.

"Tall-infra, data-only" means the entire vertical stack—from rendering to AI to audio to networking—operates on the same principle. The binary is a generic processor. The dataset is the game.

The compiled executable never learns what a dragon is. It runs state machines, evaluates predicates, scores utilities, applies envelopes. Every frame, the dataset teaches it what exists and how everything behaves.

This is not theoretical. It runs at 60fps with 20,000 transforms per frame. It scales to 10,000+ entities. It debugs through structured traces. It enables designers to create mechanics by wiring tables.

The wall can be removed. This paper describes how.

Appendix: Glossary

Entity: Universal container struct with optional capability systems. Not a type—a bag of data references.

State Machine: Graph of named states with conditional transitions. Contains no behavior code.

Fact Set: Collection of predicates true for an entity this frame. Regenerated each update.

Rule: Predicate logic expression (head :- body) evaluated against fact sets.

Consideration: Single input → normalized score via curve. Building block of utility AI.

Behavior: Collection of considerations multiplied together. Candidate for selection.

Behavior Set: Collection of behaviors. Highest score wins.

Logic Block: Stack-based bytecode for data mutation. Control flow and arithmetic.

Envelope: Time-bounded stat modifier with curve. Automated effect processing.

Trace: Structured log of decisions per entity per frame. Domain-aware debugging.

memdb: In-RAM database with O(1) column access. SQLite for persistence.

References

[**HOWL-COMP-2-2026**] Tall-Infra, Data-Only Development. Github: https://github.com/ghowland/cks/tree/main/papers/COMP-2-2026

[**HOWL-COMP-1-2026**] Implementing a Tall-Infra Data-Only Execution System . Github: https://github.com/ghowland/cks/tree/main/papers/COMP/HOWL-COMP-1-2026

[**HOWL-COMP-3-2026**] Silo OS. Github: https://github.com/ghowland/cks/tree/main/papers/COMP/HOWL-COMP-3-2026

[**HOWL-COMP-4-2026**] Geometric Security. Github: https://github.com/ghowland/cks/tree/main/papers/COMP/HOWL-COMP-4-2026

[**HOWL-COMP-5-2026**] Partial Geometric Security . Github: https://github.com/ghowland/cks/tree/main/papers/COMP/HOWL-COMP-5-2026