Living City Platform Stack Outline

Scalable Architecture for Little Rock First, Many Cities Later

bob_the_bot

06/25/2026 08:18 PM CDT

Contents

1 Executive Summary

If the long-term ambition is a believable, explorable Little Rock that can eventually expand to every major city in the United States and later internationally, the product must be designed as a **city-generation and simulation platform**, not as a one-off handcrafted map.

The safest long-range stack choice is:

- **Client and world runtime:** `Unreal Engine 5` in `C++`, with selective `Blueprint` use
- **Large-world runtime:** `World Partition`, `Data Layers`, `Hierarchical LOD`, `Replication Graph`, and `Iris`
- **Crowd and ambient AI:** `MassEntity` plus a custom simulation layer
- **Geospatial pipeline:** `Python` + `GDAL` + `GeoPandas` + `Shapely` + `PostGIS`
- **Procedural city generation:** `Houdini` for authored procedural tooling plus Unreal `PCG` for in-engine placement
- **Backend services:** `Go` for production services, with event-driven messaging
- **Core databases:** `PostgreSQL/PostGIS`, `Redis`, object storage, and `ClickHouse`
- **Live multiplayer hosting:** Unreal dedicated servers orchestrated in containers, growing into `Kubernetes + Agones`

The key architectural decision is this:

> Do not build Little Rock as a bespoke environment. Build a canonical geospatial pipeline and procedural rule system that can ingest Little Rock, then repeat for Dallas, Atlanta, London, or Sao Paulo without changing the foundational tech.

If that principle is respected, the engine choice is not the main risk. The main risks are content density, simulation scope, networking assumptions, and geospatial data normalization across regions.

2 Bottom-Line Recommendation

2.1 Recommended Long-Range Stack

2.1.1 Player-Facing Runtime

- `Unreal Engine 5`
- `C++` for systems, networking, streaming, AI runtime, save/state sync, and performance-sensitive code
- `Blueprints` only for thin gameplay scripting, UI glue, and rapid iteration
- `UMG/Common UI` for interface
- `Chaos Vehicles` only if vehicle handling needs to ship early; otherwise use a more specialized vehicle solution later

2.1.2 World and Content Runtime

- `World Partition`
- `Data Layers`
- `HLOD`
- `Nanite` where appropriate for authored hero assets
- Unreal `PCG` framework for runtime decoration and placement
- `Houdini Engine` for deterministic procedural content generation

2.1.3 Simulation Runtime

- `MassEntity` for large ambient crowds and background agent logic
- Custom layered AI model:
  - full simulation for nearby important agents
  - reduced simulation for district-level agents
  - statistical/offscreen simulation for most of the population

2.1.4 Multiplayer Runtime

- Unreal dedicated servers
- `Replication Graph`
- `Iris` replication
- region and district sharding rather than a single seamless national world

2.1.5 Geospatial and Content Build Pipeline

- `Python`
- `GDAL/OGR`
- `GeoPandas`
- `Shapely`
- `pyproj`
- `PostGIS`
- vector-tile and terrain preprocessing

2.1.6 Online Platform and Services

- `Go` for backend services
- `gRPC` for service-to-service communication
- `REST` for tool-facing and admin APIs
- `NATS` or `Kafka` for event streams
- `Redis` for cache, matchmaking/session state, and fast ephemeral reads
- `PostgreSQL` for transactional data
- `ClickHouse` for telemetry and analytics

2.1.7 Infrastructure

- `Docker`
- `Kubernetes`
- `Agones` for game server orchestration once concurrency justifies it
- `S3-compatible object storage` for world artifacts, patches, builds, and derived map assets
- `OpenTelemetry`, `Prometheus`, and `Grafana`

2.2 Why This Stack

- `Unreal` is the strongest fit for an explorable, presentation-heavy urban world.
- `PostGIS` is the safest canonical home for city data if the ambition is national or international.
- `Python` remains the best first language for geospatial ETL and procedural content preprocessing.
- `Go` is a strong balance of performance, deployment simplicity, tooling maturity, and hiring viability for backend services.
- `Houdini + Unreal PCG` gives both deterministic generation and in-engine iteration without forcing all procedural intelligence into one brittle layer.

3 What You Are Actually Building

This is not only a game. It is at least five products stacked together:

1. A **geospatial ingestion system**
2. A **procedural city compiler**
3. A **simulation platform**
4. A **multiplayer game runtime**
5. A **content and operations pipeline**

Most teams underestimate the first two and over-focus on the game client. That is how they end up with a beautiful first city and no economical path to the next ten.

4 Design Principles That Keep You Out of Trouble

4.1 1. Canonical City Schema First

Every incoming city should be normalized into one internal schema for:

- roads
- intersections
- lanes
- parcels
- zoning
- building footprints
- building classes
- terrain
- water
- vegetation
- parking
- transit nodes
- pedestrian graph
- points of interest
- spawn anchors
- district boundaries

Do not let each source dataset invent its own meaning at runtime.

4.2 2. Multi-Resolution Simulation

Do not fully simulate every person, car, business, and interior at all times.

Use three layers:

- **hero layer:** nearby player-visible agents with full behavior
- **district layer:** medium-fidelity scheduling, traffic flow, and occupancy logic
- **statistical layer:** aggregate simulation for the rest of the city

This is the difference between a product that scales and one that catches fire after two cities.

4.3 3. Exterior-First World Building

For city-scale delivery, exteriors matter first. Interiors must be tiered:

- signature interiors: authored
- common interiors: procedural kits
- inaccessible interiors: facade only

Trying to make every building enterable at equal fidelity is one of the fastest ways to choose the right engine and still lose.

4.4 4. Deterministic Generation

Given the same source inputs, the city build pipeline should be reproducible.

That means:

- source versioning
- generation-rule versioning
- seed control
- build manifests
- artifact tracking

Without deterministic builds, debugging world regressions across many cities becomes miserable.

4.5 5. Shard by Region or District, Not by Fantasy

Do not promise one perfectly seamless mega-server for all players and all cities at the start.

Build around:

- city-level regions
- district-level instances or shards
- well-defined handoff boundaries

If a future design truly justifies seamless handoff at national scale, that can be evolved later. It should not be an assumption embedded in version one.

5 Why Unreal Engine 5 Wins Here

5.1 Strengths

- strongest mainstream option for visually convincing urban exploration
- mature open-world streaming tooling
- strong C++ control for low-level systems
- dedicated server support
- broad vendor and talent ecosystem
- good fit for vehicles, character controllers, world interaction, and cinematic quality

5.2 What Unreal Solves Well

- world streaming
- traversal through large environments
- visual fidelity
- hybrid authored/procedural content
- multiplayer client/runtime integration

5.3 What Unreal Does Not Solve For You

- clean geospatial ingestion
- city data normalization
- traffic realism
- social infrastructure
- anti-cheat
- economic simulation
- a scalable content factory

That is why the engine should be seen as the **front-end runtime of the world**, not the full architecture.

6 Why Not Make Another Engine the Foundation

6.1 Unity

Unity can be excellent for simulation-heavy worlds, especially if a team is deeply committed to `DOTS/ECS`. It is still not my first recommendation for this exact product because:

- the visual/runtime fit for a premium explorable city is weaker than Unreal
- large-world urban presentation will likely take more custom work
- the long-term upside is strongest when the product is more simulation-first than immersion-first

Unity is a valid choice if the true product is closer to a large-scale urban simulation with lighter visual expectations. It is a weaker choice if the product must feel like a premium explorable world.

6.2 Godot

Godot is useful for smaller teams, lower fidelity, 2D/isometric, or fast experiments. It is not the stack I would bet on for a city-scale, visually ambitious, multiplayer urban world platform with national/international growth ambitions.

6.3 Custom Engine

No. Not unless the team already has a proven engine group and a very unusual technical constraint. Building a proprietary engine would likely move effort away from the true moat, which is the city pipeline and scalable simulation.

7 The Core World Pipeline

7.1 Source Data Layers

Each city ingest should pull from versioned data sources such as:

- road network data
- terrain and elevation
- land use and zoning
- building footprints
- address points
- water bodies
- parks and vegetation
- transit lines and stops
- parking features
- parcels and administrative boundaries
- business and landmark POIs

For United States rollout, a practical blend often includes:

- `OpenStreetMap`
- state and county GIS layers
- USGS or similar terrain products
- local parcel and zoning datasets where available

For international rollout, expect inconsistent source quality and schema differences. That is why canonical normalization is non-negotiable.

7.2 Geospatial ETL Stack

Use `Python` as the orchestration layer with:

- `GDAL/OGR` for robust geospatial conversion

- `GeoPandas` for tabular geospatial transforms
- `Shapely` for geometry operations
- `pyproj` for coordinate transforms
- `PostGIS` as the system of record for canonicalized city geometry

This stack should produce:

- cleaned road graph
- lane metadata
- terrain mesh inputs
- district boundaries
- parcel/building classes
- spawn graphs
- procedural generation descriptors

7.3 Canonical Storage Model

`PostGIS` should store the normalized city model and source provenance.

Recommended logical entities:

- `city`
- `district`
- `parcel`
- `road_segment`
- `intersection`
- `lane_segment`
- `sidewalk_segment`
- `building_footprint`
- `building_instance_seed`
- `poi`
- `transit_stop`
- `terrain_tile`
- `water_feature`
- `vegetation_zone`
- `spawn_anchor`
- `navigation_zone`
- `rule_override`

Each should support:

- source identifier
- import timestamp
- source version
- normalized schema version
- confidence flags
- manual override flags

7.4 Build Artifacts

The geospatial pipeline should emit versioned artifacts such as:

- Unreal import-ready tiles
- procedural generation descriptors
- HLOD grouping data
- navigation graphs
- AI spawn zones
- traffic seeds
- district metadata

- low-detail far-view meshes

These should be treated like compiled assets, not ad hoc exports.

8 Procedural Generation Strategy

8.1 Recommended Split

Use `Houdini` for heavy deterministic procedural generation and `Unreal PCG` for local decoration and runtime variation.

8.1.1 Houdini Should Handle

- road-adjacent lot logic
- block decomposition
- building massing rules
- facade family assignment
- roof variation
- parking lot generation
- district-level vegetation logic
- export of structured placement data

8.1.2 Unreal PCG Should Handle

- small clutter placement
- decal and prop scattering
- local foliage
- district flavor variation
- runtime density adjustments

This keeps the offline city compiler powerful while still allowing in-engine art direction.

8.2 Building Strategy

Create a tiered building system:

- `Tier 1`: hero landmarks, authored
- `Tier 2`: important commercial/residential kits, procedurally assembled from modular sets
- `Tier 3`: facade/background stock buildings

The mistake to avoid is trying to give every structure unique authored love. The right move is to concentrate authored quality where players will notice it most.

9 World Streaming and Spatial Runtime

9.1 Recommended Model

For Little Rock and future cities:

- use `World Partition`
- divide the world into streamable cells
- use district-level `Data Layers`
- use HLOD aggressively for mid and far distance
- keep interior layers separate from exterior layers

9.2 Memory and Performance Discipline

Plan around strict budgets for:

- draw calls
- actor counts
- replicated entities
- navigation volumes
- traffic agents
- pedestrian agents
- streaming IO
- texture residency

The wrong stack is not always the wrong engine. Often it is the right engine plus a team that refuses budgets.

9.3 Floating Origin and Precision

For city-scale worlds, Unreal can handle the footprint with proper large-world management. For future many-city coverage, each city or region should still be treated as a bounded operational space. Do not anchor the first architecture on the assumption of one infinitely precise always-loaded planet.

10 NPC, Traffic, and City Life

10.1 The Goal

The city must feel alive without requiring every agent to think at movie quality every frame.

10.2 Recommended NPC Model

Use layered agent fidelity:

- **hero NPCs:** full behavior trees, animation state, conversation hooks, mission relevance
- **ambient near-field NPCs:** lighter behavior with believable local goals
- **far-field NPCs:** schedule and occupancy state only

Maintain:

- home district
- work district
- daily schedule archetype
- social tag or role
- density eligibility by time/day/weather/event

Most citizens should not exist as fully instantiated pawns until needed.

10.3 Recommended Traffic Model

Traffic is often where city dreams quietly die.

Use a layered model:

- macro traffic demand by district pair
- route generation on the road graph
- local near-player vehicle realization
- statistical far-field counts elsewhere

This avoids simulating every car in the metro area at once while still making roads feel plausibly busy.

10.4 Event and Density Systems

To avoid dead-city syndrome, drive population and traffic with:

- time of day
- day of week
- district type
- weather
- venue schedules
- school/work commuting windows
- live event overrides

The feeling of reality often comes from controlled density choreography, not raw simulation purity.

11 Multiplayer and Live Player Architecture

11.1 Fundamental Rule

Do not start with the assumption that one server process should own the entire city and every player.

11.2 Recommended Model

- Unreal dedicated servers for active world spaces
- district or region shards
- handoff rules for player movement across shards when needed
- server-authoritative movement and critical state
- platform services outside Unreal for identity, economy, social, chat history, telemetry, moderation, and progression

11.3 Networking Features to Use

- `Replication Graph`
- `Iris`
- relevance filtering
- interest management by location and visibility
- network dormancy for inactive world actors

11.4 Backend Service Split

Keep these outside the game server:

- auth and accounts
- friends/social graph
- party/match/session orchestration
- entitlements and inventory
- progression
- economy
- moderation and sanctions
- telemetry and analytics
- notifications
- live-ops controls

11.5 Orchestration

Long-term, use:

- containerized dedicated servers
- `Kubernetes`

- `Agones`
- autoscaling based on population and region pressure

Short-term, do not overbuild before concurrency proves it necessary.

12 Backend and Data Platform

12.1 Primary Recommendation

Use `Go` for backend services.

Why:

- strong concurrency model
- good operational simplicity
- mature gRPC support
- good cloud/container fit
- easier hiring and debugging path than overcommitting to a more exotic backend stack

12.2 Core Service Categories

- identity/auth
- account profile
- social graph
- group/session management
- world directory
- shard allocator
- economy
- persistence gateway
- moderation
- content/version manifest
- telemetry intake
- admin tools

12.3 Data Layer Recommendation

12.3.1 PostgreSQL / PostGIS

Use for:

- core transactional data
- world metadata
- canonical city geometry
- rule tables
- versioning metadata

12.3.2 Redis

Use for:

- hot cache
- matchmaking/session lookups
- ephemeral presence
- distributed locks where justified

12.3.3 ClickHouse

Use for:

- analytics
- world heatmaps
- pathing stats
- player movement analysis
- funnel analysis
- performance telemetry

12.3.4 Object Storage

Use for:

- derived world artifacts
- build outputs
- map bundles
- logs
- asset manifests
- replay or audit blobs if needed

12.4 Messaging Layer

Use `NATS` if you want simpler operational posture early.

Use `Kafka` if event volume, durable streams, and analytics fan-out become central earlier than expected.

Default recommendation:

- start with `NATS`
- move selective pipelines to `Kafka` only when clearly justified

13 Tooling and Content Production

13.1 Editor and Runtime Code Policy

- `C++` for engine-side systems
- `Blueprint` for controlled high-level logic only
- strict code ownership around networking, AI runtime, and streaming

If Blueprints become the main systems layer, the project will likely become harder to scale and reason about.

13.2 Art Pipeline

Recommended tools:

- `Houdini`
- `Blender`
- `Substance 3D` or equivalent texturing pipeline
- Unreal build automation

Use modular kits by district archetype:

- downtown
- suburban residential
- strip commercial
- industrial
- civic/institutional
- parks/recreation

13.3 Versioning and Build Discipline

- `Git` for code and small config
- `Perforce` is worth serious consideration for large Unreal content teams
- build manifests for every city compile
- asset provenance tracking
- reproducible city builds

If the team remains small for a long time, Git can work. If asset volume and team size grow materially, Perforce becomes much more attractive for Unreal production.

14 Observability, QA, and Performance Governance

14.1 You Need More Than Crash Logs

Instrument:

- frame time
- streaming stalls
- replication cost
- AI update cost
- server tick time
- district density
- vehicle counts
- spawn/despawn churn
- player travel paths
- hotspot districts

14.2 Recommended Stack

- `OpenTelemetry`
- `Prometheus`
- `Grafana`
- centralized logs
- build dashboards per city and per shard

14.3 Mandatory Test Types

- world build reproducibility tests
- map data schema validation
- streaming boundary tests
- multiplayer soak tests
- district handoff tests
- traffic load tests
- spawn density regression tests
- city compile duration regression tests

15 Security and Trust Boundaries

15.1 Do Not Trust the Client

Anything that matters for fairness, economy, or persistence should be server authoritative.

15.2 Protect the Pipeline

Because the true moat is likely the city compiler and data pipeline:

- sign build artifacts
- protect source datasets and transforms
- separate public game clients from internal generation tools
- log admin overrides
- restrict live-ops powers tightly

16 Cost and Team Implications

16.1 This Stack Is Powerful, Not Cheap

The good news:

- it keeps future optionality high
- it preserves a path from one city to many
- it avoids locking the whole product to a weak visual stack

The bad news:

- it requires real technical leadership
- it requires discipline around simulation scope
- it still needs careful staged rollout

16.2 Team Shape to Support It

At minimum, the architecture eventually wants strong ownership in:

- Unreal gameplay/runtime engineering
- networking/server engineering
- geospatial/data engineering
- procedural tools/content engineering
- technical art
- backend/platform engineering
- QA/performance engineering

One generalist can prototype this. A production platform cannot stay a one-person magic trick forever.

17 What to Avoid

17.1 Avoid These Strategic Mistakes

- hand-building the first city too deeply before the pipeline exists
- promising every building is enterable
- assuming full-fidelity simulation everywhere
- keeping all multiplayer authority inside one giant Unreal server
- letting raw GIS source quirks leak directly into gameplay/runtime data
- overusing Blueprints for core systems
- choosing tools that are easy for city one but brittle for city twenty
- locking core world generation to a proprietary third-party data service without an exit path

18 Recommended Phased Rollout

18.1 Phase 0: Architecture and Pipeline Prototype

Goal:

- prove source ingestion
- prove canonical schema

- prove one district can compile into Unreal reliably

Deliverables:

- Little Rock pilot ingest
- canonical city schema
- procedural building and road rules
- walkable district prototype

18.2 Phase 1: Vertical Slice

Goal:

- one dense, convincing district with vehicles, pedestrians, and a small multiplayer population

Deliverables:

- streaming runtime
- NPC fidelity layers
- district traffic model
- dedicated server prototype
- tooling for repeated district rebuilds

18.3 Phase 2: Full Little Rock Playable Exterior

Goal:

- full metro footprint within the chosen radius at sensible exterior fidelity

Deliverables:

- district partitioning
- HLOD strategy
- broader traffic patterns
- hero landmarks
- live population tuning

18.4 Phase 3: Second and Third Cities

Goal:

- prove portability of the pipeline

Deliverables:

- ingest adapters for new cities
- rule portability report
- updated district archetype library
- compile-time and runtime cost comparisons

This is the real stack test. If city two is painfully custom again, the platform is not ready.

18.5 Phase 4: Multi-City Live Operations

Goal:

- support multiple live cities with stable operations and content refresh

Deliverables:

- shard management
- deployment orchestration
- analytics dashboards

- live-ops tooling
- asset/content release governance

19 Final Verdict

If I were trying to minimize the chance of getting a few cities in and then discovering the whole technical foundation was wrong, I would choose this posture:

- `Unreal Engine 5` as the world client and real-time runtime
- `C++` as the core engine-side language
- `World Partition + Replication Graph + Iris + MassEntity` as the key runtime systems
- `Python + GDAL + GeoPandas + Shapely + PostGIS` as the geospatial backbone
- `Houdini + Unreal PCG` as the procedural content stack
- `Go + PostgreSQL/PostGIS + Redis + ClickHouse + NATS` as the online platform core
- containerized dedicated servers, with a path to `Kubernetes + Agones`

That combination is not the cheapest path to a flashy prototype, but it is the strongest path I see to:

- one believable city now
- many cities later
- high visual quality
- real player presence
- preserved optionality

The wrong architecture would be to optimize for short-term ease and accidentally build a handcrafted Little Rock that has to be reinvented city by city. The right architecture is to treat Little Rock as the first compiled output of a reusable urban-world platform.

20 Decision Checklist

Before locking the stack, confirm these are true:

- the product needs premium 3D city exploration, not just simulation
- exteriors matter more than universal interior access
- city data ingestion will be a strategic asset
- multiplayer should scale by regions or districts, not one giant world process
- the team is willing to invest in a real procedural and geospatial pipeline
- the company wants portability from city one to city many

If those are true, this stack is the one I would bet on.