SEAGLASS FOUNDRY

ForgeGIS Studio™

Technical Brief

The operator surface for the Forge suite —
where the engines become something an analyst uses.

Version 1.0 · July 2026

Forged For Speed. Refined by Craft.

Seaglass Foundry LLC · Panama City, Florida

Executive Summary

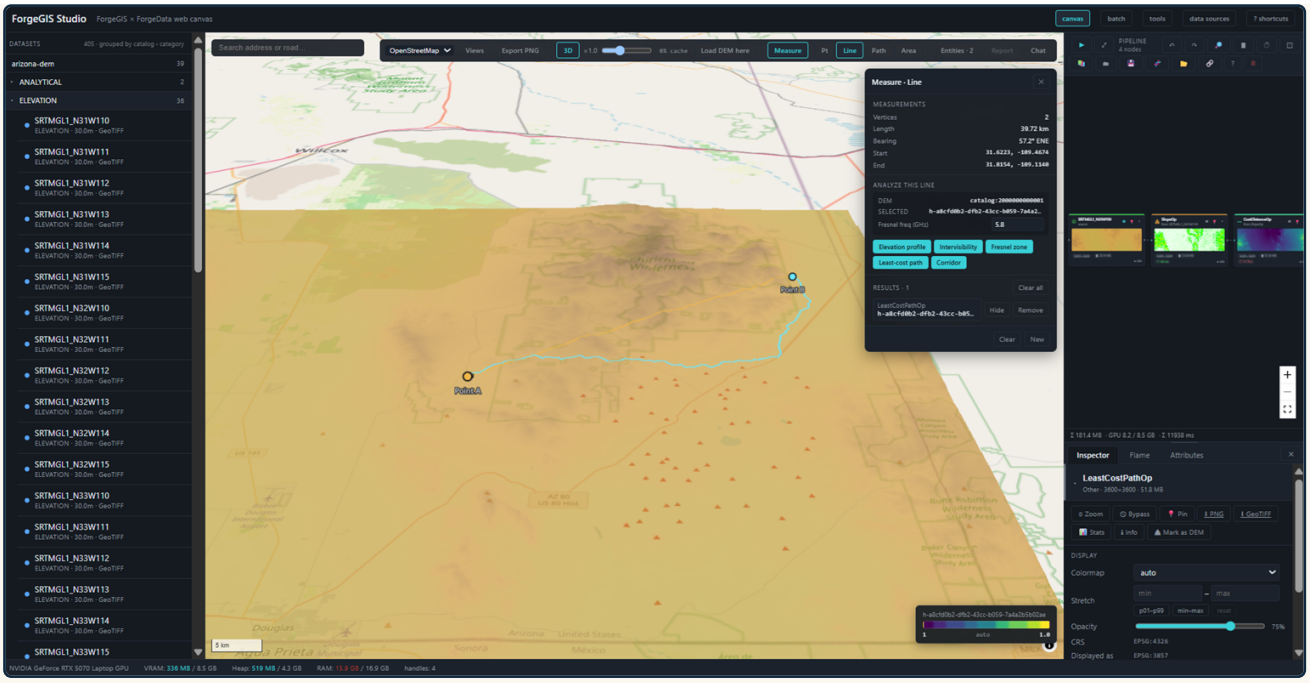
ForgeGIS Studio is where the Forge suite becomes visible. Three engines do the work — ForgeGIS (GPU-accelerated compute), ForgeData (the spatial data catalog), and ForgeMind (the natural-language control plane) — but they expose only libraries and machine protocols. Studio is the browser-based workspace that puts a map, a pipeline canvas, and a conversational agent in front of a person, so the suite's capabilities can be seen, exercised, and evaluated through one interface.

Studio is not a general-purpose GIS desktop, and it is not trying to be one. It is a demonstration and operator surface for the Forge engines: a place to show that a GPU-resident pipeline runs in real time, that raster and vector operations compose into non-linear graphs, that a natural-language agent can drive a live map, and that the whole stack runs on a single host with no network exposure. Where a customer needs a specific workflow, Studio is built to be **shaped to it quickly** — the surface is deliberately thin over the engines precisely so it can be re-cut to a customer's task without re-plumbing the compute underneath.

Three capabilities show what the suite can do through Studio. First, an interactive canvas and node editor where raster **and** vector pipelines — including mixed raster-to-vector graphs with branches and joins — are authored, run, saved to a library, and replayed. Second, a batch runner that executes those same pipelines across a folder of inputs with per-tile resource pre-flight, so a graph proven on one file runs over hundreds without a memory surprise. Third, a live bridge to ForgeMind: the agent places markers, draws overlays, and pushes reports onto the operator's own map, with those results persisted as durable, named scene entities that survive a restart.

Studio is deliberately a single-user, single-host application served in a browser, not a multi-tenant web platform. That constraint is a security decision, not a limitation: the HTTP surface binds to loopback only, path access is mediated by a sandbox, and the agent reaches the same operations through an authenticated bridge rather than a second, weaker door. The result runs on one workstation — including on a disconnected network — which is exactly the posture the suite's target environments require. That single-host model is the first configuration, not the last: a roadmap toward shared-GPU and multi-GPU deployments is described at the close of this brief, framed as a stated direction with real technical work still ahead rather than as shipping features.

This brief describes what Studio demonstrates, how it is built, how quickly it can be tailored to a customer's workflow, and the security posture that makes its surface safe for the deployment environments the Forge suite targets — including on-premises, air-gapped government networks.



The ForgeGIS Studio workspace: a least-cost path computed on a GPU-resident Arizona elevation dataset and drawn between two points on the live map. The catalog rail (left) browses ForgeData; the measure and analysis panel (right) drives ForgeGIS operations; pipeline result thumbnails and the Inspector show recent GPU outputs; the status bar reports the RTX 5070 device, VRAM, heap, and live handles.

1. What ForgeGIS Studio Is

The Forge suite is engine-first by design. ForgeGIS is a library and an MCP server. ForgeData is a catalog and an MCP server. ForgeMind is a compiler and a control plane. None of them draws a map. Studio is the component that makes the suite's capabilities tangible: a browser workspace, backed by a local server, that loads data, runs operations, shows results on a live map, and hosts the agent conversation — so what the engines do can be watched happening.

Studio's job is demonstration and operation, not feature-parity with a general GIS desktop. It does not set out to reproduce the hundreds of dialogs and decades of accreted tooling in a mature desktop GIS, and it makes no claim to. What it sets out to do is prove the things that make the Forge suite different — GPU-resident pipelines that run interactively, non-linear raster/vector graphs, a natural-language agent driving a real map, single-host deployability — and to be **re-shaped quickly** when a customer needs those capabilities delivered in the exact shape of their workflow.

The capabilities Studio surfaces:

- **A pipeline canvas and node editor.** A ReactFlow graph editor where ForgeGIS operations become nodes, handles connect them, and non-linear topologies — branches, merges, diamonds — express real analytical workflows rather than a single linear chain. This is where the engine's composability is made visible.
- **A batch runner.** The same pipelines, executed over many files with resource pre-flight, a priority queue, out-of-bounds skip handling, and a CSV run report — the proof that a graph proven once scales to a production folder.
- **A catalog browser.** Direct access to ForgeData's registered datasets, with AOI-windowed reads so an analyst pulls just the extent they need rather than a whole continental tile.
- **A ForgeMind chat rail and agent bridge.** A conversation surface where the natural-language agent answers questions, runs read operations, and — through the authenticated bridge — acts on the operator's own map.
- **A live map.** A MapLibre canvas with 2D and 3D terrain viewing that renders every result the engines produce, so the output of a pipeline or an agent turn is seen immediately, not exported and opened elsewhere.

Studio is not sold separately. It is the operator surface of the Forge suite, and its value is realized with the engines behind it. It is described here as its own component because what it contributes — a place to see the suite work, and a surface that bends to the customer — is distinct from what the engines contribute.

2. Architecture

Studio is a thin, local, three-tier application: a React frontend in the browser, a Java HTTP/WebSocket server on the same machine, and the Forge engines reached in-process or over MCP.

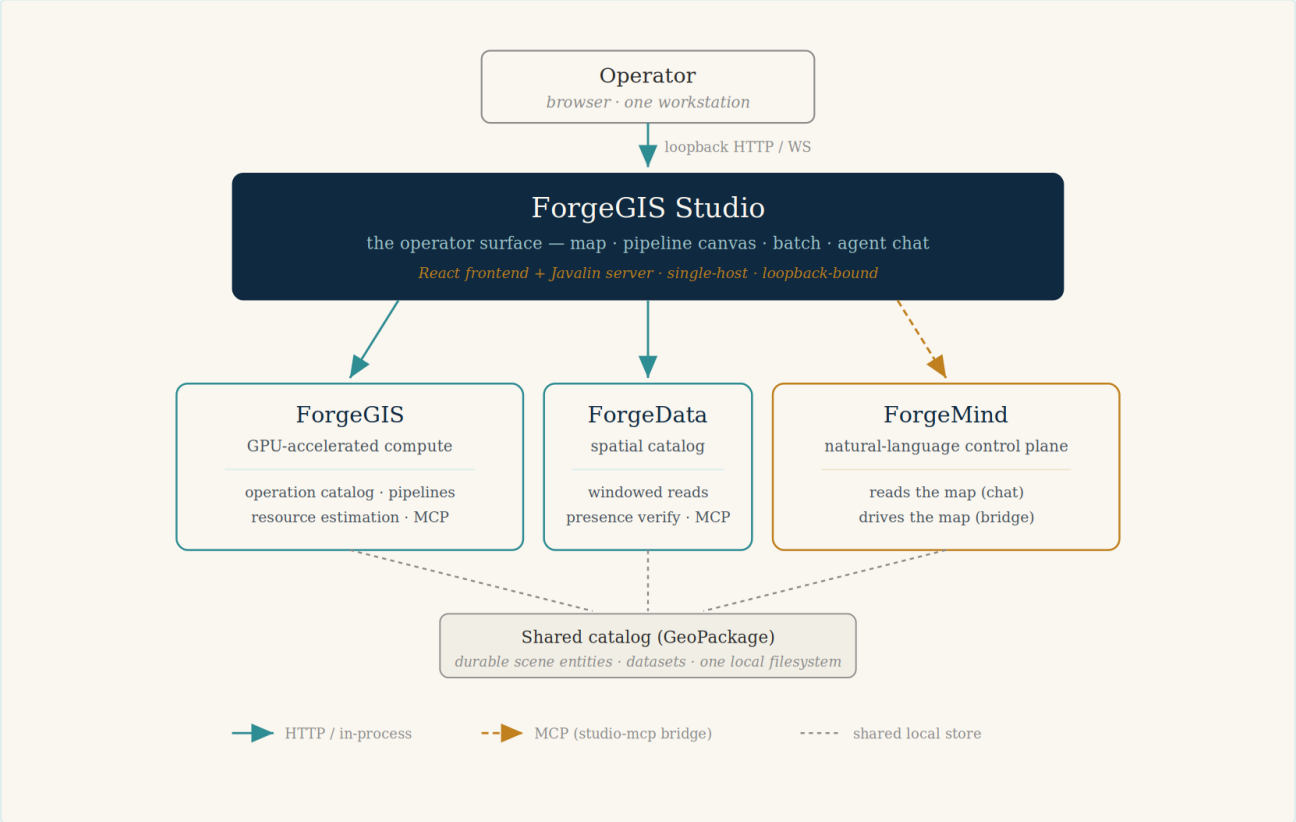


Figure 1 — Studio sits above the three Forge engines as the one operator-facing surface, reached only from the local host's browser. The engines share a single local catalog.

Frontend. A React single-page application using MapLibre GL for the map and ReactFlow for the pipeline graph. It talks to the local server over a small HTTP API and receives live status — backend device, VRAM, heap, system RAM, and job progress — over a WebSocket.

Server. A Javalin (Jetty) HTTP server bound to IPv4 loopback. It exposes a compact `/api/*` surface — operations catalog, pipeline apply, raster and vector transport, catalog reads, batch jobs, filesystem roots — plus a `/ws` WebSocket for status events. The server holds the session state, the op executor pool, and the handles for rasters and vectors that pipelines operate on.

Engines. ForgeGIS operations run through the shared op catalog, which unions the raster, vector, and cross-type registries and stamps each operation with its kind and input ports. ForgeData is reached through a narrow facade over its service, giving Studio catalog listing, AOI-windowed raster extraction, and dataset presence verification. ForgeMind connects through the MCP bridge described in §5.

The design keeps three properties deliberately true:

- **Loopback-bound.** The server listens on `127.0.0.1` by default. Other hosts on the same network cannot reach it unless an operator explicitly rebinds outward — and if they do, Studio warns on every boot until hosted-deployment prerequisites are met.
- **In-process where it matters.** ForgeGIS operations execute against the same GPU backend the server already holds. There is no second process to marshal a raster across for an interactive apply.
- **One session, many surfaces.** The browser, the batch runner, and the agent bridge all act within the same session, over the same handles. The agent draws on the map the operator is looking at — not a copy.

3. The Pipeline Canvas

Studio's node editor is where the suite's compute capability becomes composable by hand. Operations from the ForgeGIS catalog appear as nodes; connecting their handles builds a pipeline; the topology can branch and merge rather than run as a single line.

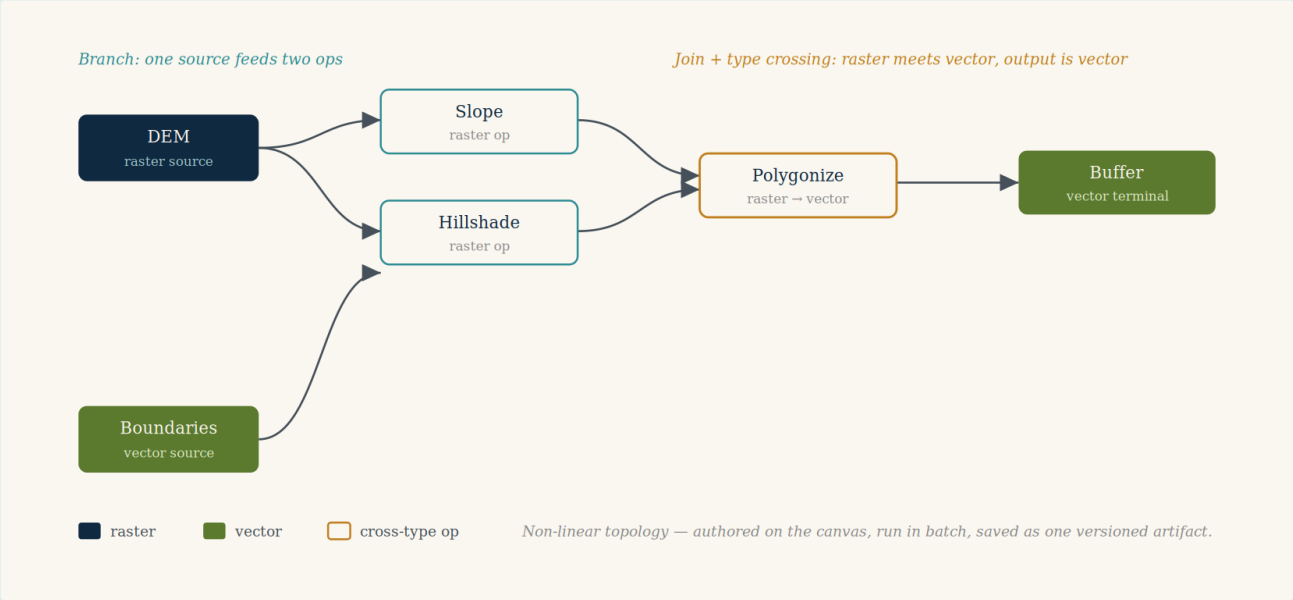


Figure 2 — A non-linear graph mixing raster and vector operations. One source branches to two operations; a raster-to-vector step crosses types; the terminal is vector. The whole graph is one saved, replayable artifact.

Raster, vector, and mixed graphs. Studio authors and runs raster pipelines, vector pipelines, and mixed raster-to-vector pipelines on the canvas. A `Polygonize → Buffer` chain that crosses from raster into vector is expressed directly: the palette gates operations by the selected node's kind — vector operations off a vector node, a raster-to-vector operation off a raster node — so the graph the operator builds is always type-valid. Vector geometry is transported to the canvas for preview through a dedicated endpoint, so a vector node draws its result the same way a raster node shows its output.

A saved-pipeline format that carries topology. Pipelines serialize to a versioned format that records the pipeline type, each operation's kind, its inputs, and its dependencies — enough to reconstruct a non-linear graph faithfully. A pipeline authored on the canvas saves to a library and replays kind-aware, with genuine non-linearity detected from the graph's structure rather than assumed. The same format is what the batch runner consumes, so there is one pipeline representation across interactive and batch execution, not two.

DAG as a capability, not a benchmark. The value of the node editor is topology — the ability to express joins, diamonds, and branches across raster and vector data in one graph — not raw speed. Speed comes from the engine underneath. The canvas is how an operator reaches for that speed in a shape that matches the analysis they actually need to run.

4. The Batch Runner

A pipeline proven on one file is worth little if running it over a folder of inputs is a separate, hand-built exercise. Studio's batch runner closes that gap: it executes the same saved pipelines — raster, vector, and mixed — across many inputs, writing one output per input in the terminal type the pipeline produces.

Resource pre-flight, not trial and error. Before committing a batch, Studio estimates each tile's peak VRAM through the shared resource-estimation contract exposed by ForgeGIS, and gates the run against the available budget rather than discovering the ceiling by crashing. The estimate is arithmetic, derived from the operation catalog's cost descriptors — not a sample run of the first input. Interactive applies use the same estimator, so a pipeline the batch runner accepts is a pipeline the interactive canvas would also accept; there is one memory ceiling across both, not two that disagree.

Honest capacity reporting. Studio's status bar reports the backend device, its VRAM budget and current use, JVM heap, and system RAM, shifting from green to amber to red as the budget tightens. Where the underlying compute backend cannot report free memory directly, Studio reports the honest derived figure rather than a comforting but meaningless one.

Built to run unattended. The runner carries a priority queue, wraps each operation in progress events, skips inputs that fall outside a raster's bounds rather than failing the whole batch, and writes a CSV report with estimated and actual VRAM per tile so a run can be audited after the fact. Output format is selectable where it matters — GeoPackage by default for vector terminals, GeoJSON where an operator wants it.

5. The ForgeMind Bridge

The most distinctive thing Studio does is let a natural-language agent act on the operator's own map. This is what turns ForgeMind from a system that answers questions into one that does work an analyst can see.

Two paths, one deliberately more capable. Over the plain chat rail, ForgeMind answers in text and runs read-only operations — it can inspect the viewport, query elevation in a region, and reason about what it finds. To *act* on the canvas — place markers, draw overlays, push reports — the agent connects through the `studio-mcp` bridge, an authenticated channel that exposes Studio's map as a set of MCP tools the agent can call. The distinction is intentional: reading the map is lightweight and always available; driving the map requires the bridge and its credential.

Authenticated, not open. The bridge is reached through an agent-key exchange rather than a shared token pasted into a dialog. The browser-facing API uses a lazy-minted session that is safe only under loopback; the agent bridge uses a separate, token-validated path that will not re-mint on a stale session. An earlier copy-paste token flow was removed in favor of this exchange — a deliberate reduction in the number of ways a credential can leak.

Results that persist. When the agent places a named artifact on the map — a located feature, a marked route, a labeled region — Studio records it as a durable *scene entity* in the ForgeData catalog, not as session scratch. The catalog row is the source of truth; a session is a cache over it. When Studio restarts, entities are adopted back into the new session with their identities intact, so an agent's cached handle to "the entity I placed last turn" still resolves. Transient agent overlays stay transient by design; named entities the user placed are durable. The result is an agent whose work on the map is not lost the moment the process cycles.

6. Data Access

Studio reads geospatial data through ForgeData rather than owning file I/O itself, which keeps one catalog and one extraction strategy across the suite.

Catalog-backed loads. An operator browses ForgeData's registered datasets and loads one into the session. Studio forwards the request to ForgeData and receives back a local file to read — it does not reimplement the catalog's knowledge of where data lives or how to reach remote tiers.

AOI-windowed reads. When the operator supplies an area of interest, Studio asks ForgeData for just that window rather than the full source tile. A hillshade of a canyon pulls a few thousand pixels on a side, not a continental raster — and, as a side effect of routing through the catalog's extraction layer, area-of-interest reads against remote sources that Studio could never open directly begin to work. The windowed extract is sized to fit under Studio's raster cap before it ever reaches the GPU.

Presence verification. A catalog can register more than is actually on disk. Studio consumes ForgeData's presence-verification signal so a dataset that is registered but missing is caught at pick time — with a clear message — rather than exploding at read time with an opaque file-not-found. The verification is on-demand and timestamped, so each consumer chooses its own staleness threshold rather than paying to re-stat every path on every list.

7. Security Posture

Studio's HTTP surface is intentionally light on authentication because its deployment model is a single operator on a single machine. That model is enforced, not assumed.

Loopback by default. The server binds to `127.0.0.1`. Hosts elsewhere on the LAN cannot reach it. If an operator rebinds outward, Studio detects the non-loopback bind and warns on every boot until the hosted-deployment prerequisites — real authentication, CSRF protection, rate limiting, TLS termination, an audit log, and a re-review of sandbox scoping — are in place. The warning is deliberate and is not meant to be papered over.

A mediated path perimeter. Every route that accepts a caller-controlled path runs it through a sandbox before touching the filesystem. Uploads are confined to an upload directory with a write-extension allowlist and no overwrite. Catalog reads are checked against the resolved file path, treating the catalog row — not the request body — as the dangerous input. Filesystem roots the user opts into through a picker are canonicalized, resolved through their symlinks, and rejected if they reach sensitive system locations; they are stored session-scoped, so one session's opt-in is invisible to another session and to the agent path. Agent-driven tool calls run under an operator-only sandbox that cannot reach paths a browser session added.

Bounded resources. Upload byte ceilings, total-pixel and per-side raster caps, a pre-dispatch VRAM estimate gate, a bounded op-executor pool that refuses rather than thrashes when full, and a per-operation timeout together bound the worst case a single workstation has to absorb.

A separate door for the agent. The lazy-minted browser session — safe only under loopback — is one path. The bridge and agent-key exchange are another, token-validated, path. They do not share a credential model, and the stronger one guards the ability to act on the map.

This posture is what makes Studio appropriate for the environments the Forge suite targets. On an unclassified on-premises defense network, the whole suite runs on one host, reachable only from that host's own browser, with the agent driving the map through an authenticated bridge — no network exposure, no external calls required for the core workflow.

8. Built to Be Customized

Studio is intentionally a thin surface over the Forge engines, and that thinness is what makes it fast to tailor. The compute, the catalog, and the natural-language control plane are the hard, stable parts of the suite; Studio is the layer that decides what a particular customer sees and touches. Re-cutting that layer for a specific mission does not disturb anything underneath.

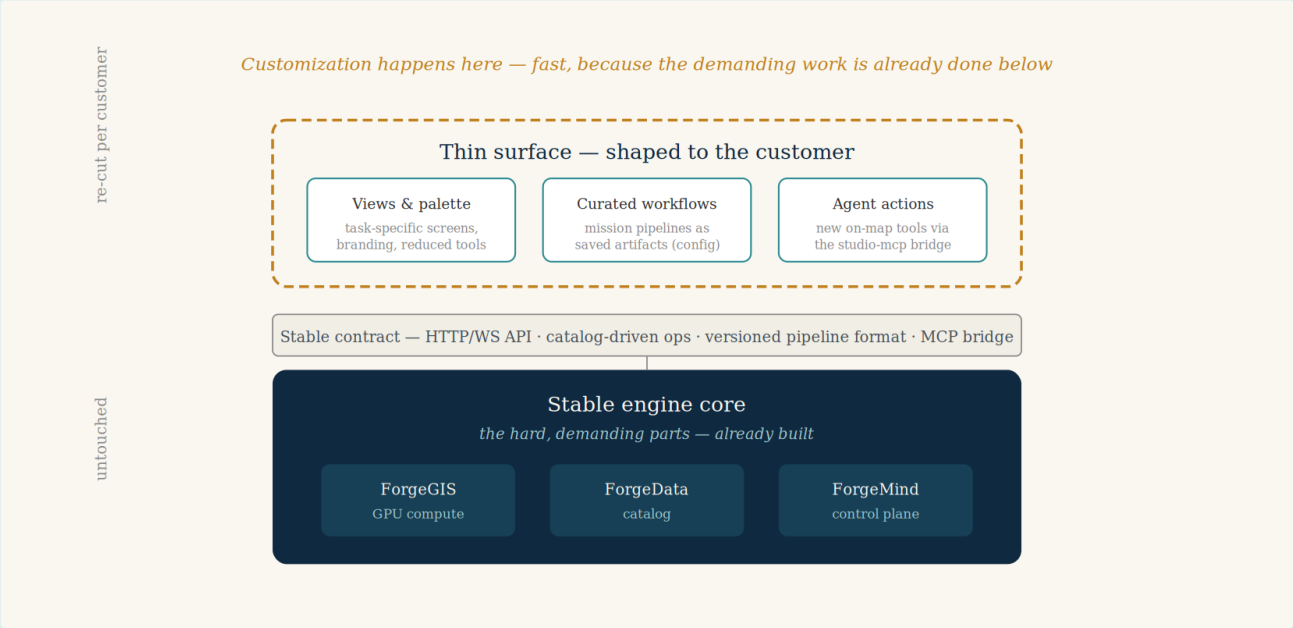


Figure 3 — The demanding engine work is built and stable. Customization happens in the thin surface above a stable contract — views, curated workflows, and agent actions re-cut per customer without touching the compute.

A small, well-defined surface. Studio's server exposes a compact HTTP API — an operations catalog, pipeline apply, raster and vector transport, catalog reads, batch jobs, filesystem roots — and a WebSocket for live status. Its frontend is a React application over that API. A customer workflow is built by composing these existing surfaces, not by reaching into the engines. New screens, a reduced or task-specific tool palette, a purpose-built operator view, or a customer's own branding are frontend work against a stable backend contract.

Capabilities flow through, they aren't re-hardcoded. Operations are not enumerated by hand in Studio; they arrive from the ForgeGIS catalog, each stamped with its kind and input ports, and render as nodes automatically. When the engine gains an operation, it appears in Studio without a Studio code change. A customer who needs a domain-specific operation gets it added to the engine once and it is immediately authorable, runnable, and batchable through the existing canvas — the customization is in the engine's capability, and Studio surfaces it for free.

Workflows are data, not code. Pipelines serialize to a versioned format that fully captures a non-linear graph — pipeline type, per-operation kind, inputs, and dependencies. A workflow tuned for a customer is a saved artifact that ships with a deployment and replays faithfully, on the canvas or in the batch runner. Curating a library of mission-specific pipelines is a configuration task, not an engineering one.

The agent surface is composable too. The `studio-mcp` bridge exposes Studio's map to ForgeMind as a set of MCP tools. Extending what the agent can do on the canvas — a new kind of overlay, a customer-specific report, a bespoke marker behavior — is adding a tool to that bridge, cleanly separated from both the engine and the browser UI.

The practical consequence: a customer engagement typically does not begin with "build a GIS application." It begins with a working suite and a working Studio, and the work is shaping the surface — the views, the palette, the curated workflows, the agent's on-map actions — to the customer's task. Because the demanding parts are already built and stable, that shaping is measured in the time it takes to compose known pieces, not the time it takes to build a platform. As a solo, craft-driven shop, Seaglass Foundry can turn that customization around directly, without the coordination overhead of a large vendor.

9. Deployment Today

Studio is a Java server plus a static frontend, packaged to run on a single host. It starts from a run script, resolves a ForgeData catalog and its datasets, and serves the browser workspace on a local port. It is designed to sit at the top of the suite's deployment stack: the one externally reachable surface (on loopback) above ForgeGIS, ForgeData, and ForgeMind.

Because the whole suite is single-tenant and single-host by design *today*, Studio deploys the same way the suite does — on a workstation, on an on-premises server, or inside a single-tenant container — without a multi-tenant control plane to stand up first. The most likely first deployment environment for the suite is an unclassified on-premises government network, and Studio's loopback-bound, air-gap-friendly posture is built for exactly that. This is a deliberate starting point: a self-contained, honest, fully-working configuration that an evaluator can run end-to-end on one machine — the foundation the configurations below build on, not a ceiling.

10. Where Deployment Is Headed

The single-host model is the first configuration, not the only intended one. GPU compute is expensive and often shared, and real programs have more than one analyst. The direction of travel is to let the same suite serve more than one operator and draw on more than one accelerator, without abandoning the security discipline that makes the current model safe. Three configurations describe the vision:

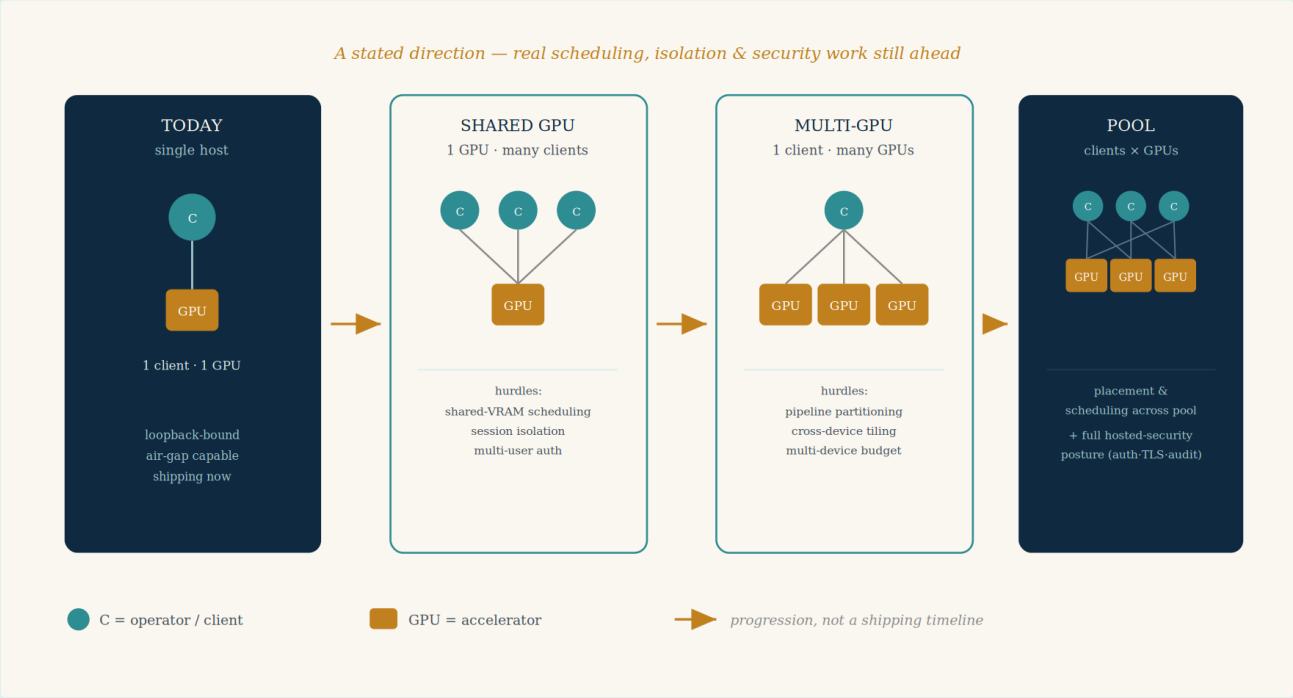


Figure 4 — The deployment roadmap. Today's single-host configuration is the foundation; shared-GPU, multi-GPU, and full pool configurations are a stated direction, each with its own scheduling, isolation, and security work still to be done.

- **One GPU, multiple clients.** A single accelerator-equipped host serving several operators — the natural fit for a team sharing one workstation-class or server-class GPU. The hard problems here are fair scheduling of a shared VRAM budget across concurrent sessions, isolation between operators, and moving past the lazy-mint loopback session model to real multi-user authentication.
- **One client, multiple GPUs.** A single operator's heavy pipeline spread across several accelerators on one host — for the large-area, high-resolution work where one GPU's memory is the limit. The hard problems here are partitioning a pipeline (and its tiles) across devices, and reconciling the resource-estimation contract with a multi-device budget.
- **Multiple clients, multiple GPUs.** The general case: a pool of operators drawing on a pool of accelerators, with work placed where capacity exists. This composes the first two and adds the genuinely hard parts — placement and scheduling across a device pool, and the full hosted-deployment security posture (authentication, CSRF protection, rate limiting, TLS, and audit) that Studio's current design already names as the prerequisites for any non-loopback bind.

These are a stated direction, not shipping features. Substantial technical work remains in scheduling, isolation, cross-device execution, and the security hardening that off-host deployment requires — the security posture section of this brief already enumerates those prerequisites deliberately, precisely so that scaling outward is a planned progression rather than a shortcut. What makes the progression credible is the architecture Studio already has: a thin surface over engines that own their own resource accounting, a resource-estimation contract shared across interactive and batch execution, and a security model that is explicit about exactly what must be added before the port leaves the loopback. The foundation was built to be grown from.

11. Specifications at a Glance

Property	Value
Role	Operator surface for the Forge suite
Frontend	React single-page app; MapLibre GL map; ReactFlow node editor
Server	Javalin (Jetty) HTTP + WebSocket, Java 17+
Bind	IPv4 loopback (`127.0.0.1`) by default
Pipeline types	Raster, vector, mixed (raster ↔ vector); non-linear DAG topology
Batch	Multi-input execution, resource pre-flight, priority queue, CSV report
Agent surface	ForgeMind chat rail + authenticated `studio-mcp` bridge
Scene persistence	Durable scene entities backed by the ForgeData catalog
Data access	ForgeData catalog reads, AOI-windowed extraction, presence verification
Resource safety	Shared ForgeGIS resource-estimation contract; upload/pixel/dimension caps
Customization	Thin surface over stable engines; catalog-driven ops, versioned workflow artifacts, composable agent bridge — rapid tailoring to customer workflows
Deployment (today)	Single-user, single-host; on-premises / air-gap capable
Deployment (roadmap)	Shared-GPU multi-client, multi-GPU single-client, and multi-client/multi-GPU pools — a stated direction, with scheduling, isolation, and hosted-security work still ahead

About Seaglass Foundry

Seaglass Foundry LLC is an independent geospatial software company based in Panama City, Florida, building the compute substrate for AI-native geospatial analysis. Its product line includes ForgeGIS (GPU-accelerated geospatial library and MCP server), ForgeData (spatial data catalog), ForgeMind (LLM-agnostic natural-language control plane), ForgeGIS Studio (the operator surface described here), and SwingToPDF (vector PDF export for Java Swing applications, in production).

For evaluation, licensing, or partnership inquiries: **rich@seaglassfoundry.com** · seaglassfoundry.com