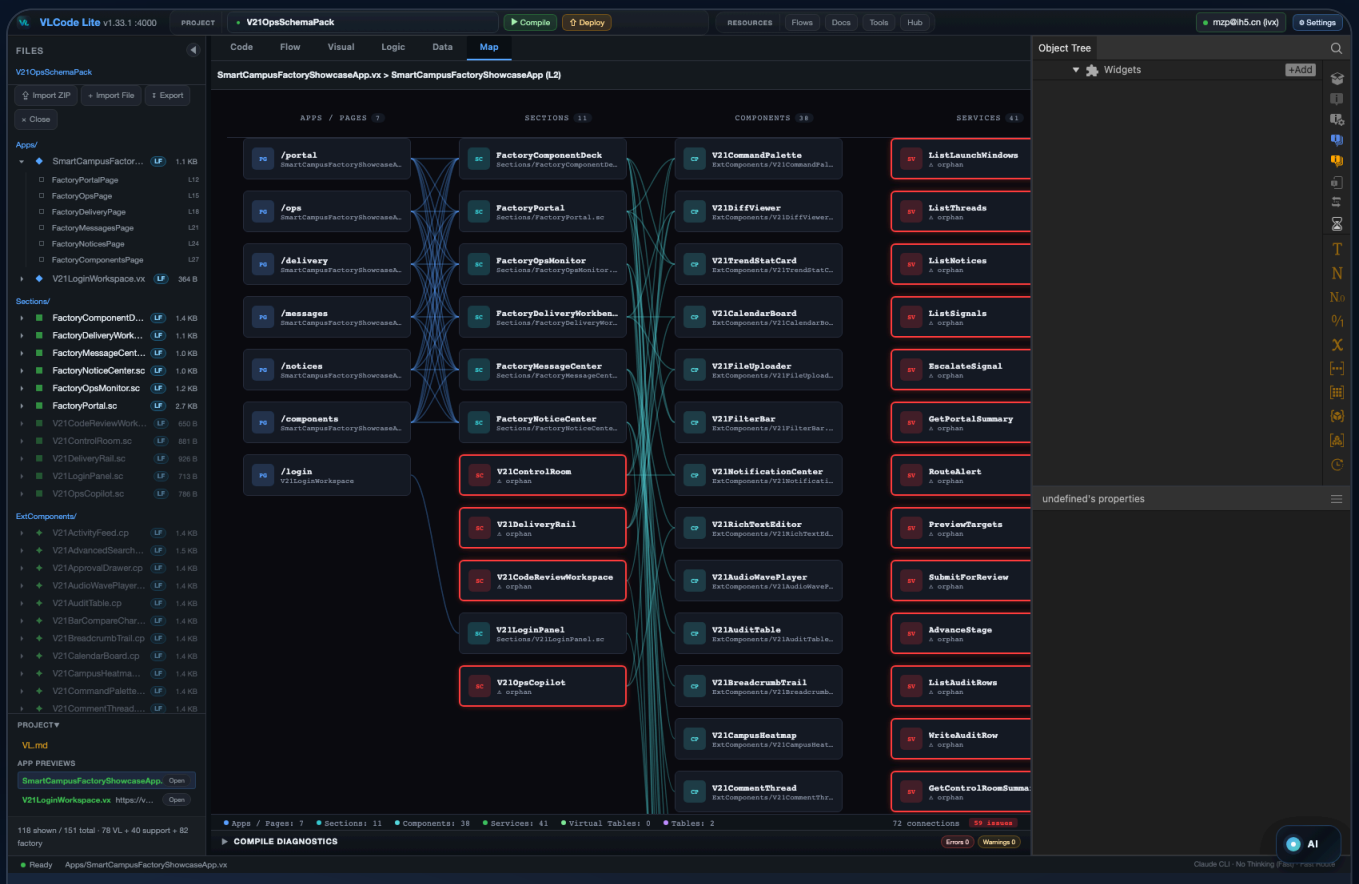
VisualLogic

AI-NATIVE DEVELOPMENT PLATFORM

Tame AI inside a DAG.

Today's AI writes brilliant **points**. VLC turns them into a **line you can trust** — keeping the model's capability and creativity, while the graph keeps your execution logic stable.



Map mode — an entire VL application rendered as a single architecture graph.

VL-Code Open source (MIT) · Local-first **github.com/VisualLogic-AI/VL-Code**

01 · THE CORE IDEA

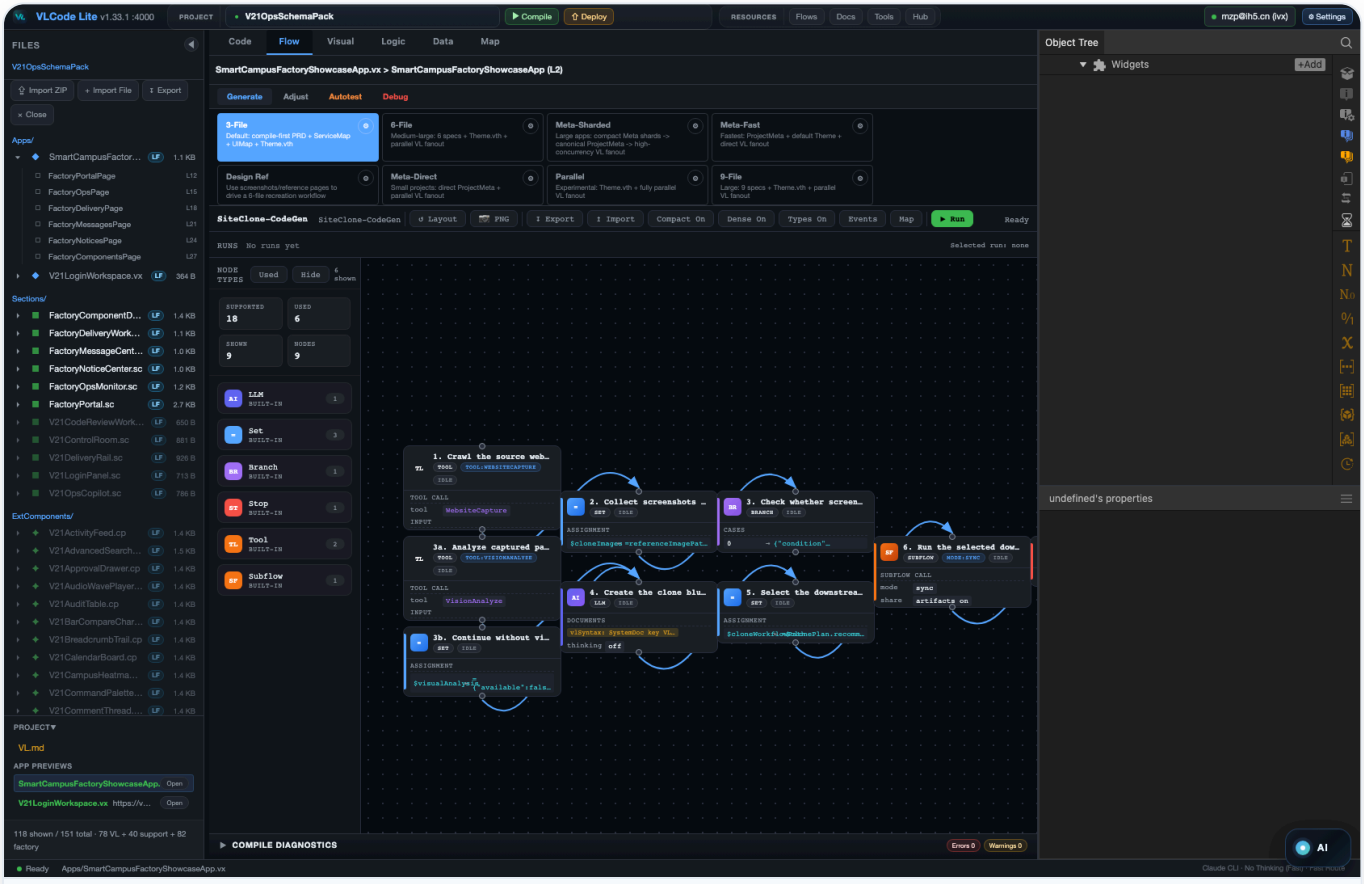
AI is brilliant at a point. It can't hold a line.

So VLC puts the AI inside a graph.

Today's best coding agents — Claude Code, Codex — are extraordinary. But they operate on a **point**: one input, one output. As a developer you can't predict that output, which means you can't predict the *next step's starting point*. So you must watch every step and add real value at each one. That's a heavy demand — and it means you were never really set free.

> **The real question isn't "a smarter model." It's: how do you keep the capability and creativity of AI and a stable execution logic — at the same time?**

VLC's answer is to place the AI inside a **DAG**. Creativity lives **inside each node**; determinism lives **in the edges between them**. The model is free to be brilliant at every step, while the graph guarantees the step after it lands exactly where you expect. You stop babysitting and start orchestrating.



Flow mode — every build runs as a verifiable DAG: generate → check → fix → preview.

02 · THE LEVELER

Same AI. A 10× gap between people.

A DAG is what flattens it.

On a single **point** — one input, one output, however rich — you can no longer out-think the model. That race is over. But real applications were never a point: they're a **line** of many points, stacked in layers. The whole job now is to *drive* every input and output along that line.

> **Facing the very same AI, one builder ships ten times the value — and another is left helpless. The difference isn't the model. It's whether you can hold the line.**

That's exactly what a **DAG** does: it is the path-plan that drives the AI across the whole line, node by node — and the AI can draw that plan itself. The gap between people collapses, because it's the structure, not the person, that carries the line.

GIVEN 1

Logic decomposes — and recomposes

Any complex process splits cleanly into points and edges; the compute behind it splits the same way. What can be decomposed can be orchestrated.

GIVEN 2

AI is exceptional at logic

Hand the model explicit structure and it plans, fills and checks each step reliably — far more so than when it must hold an entire program in its head.

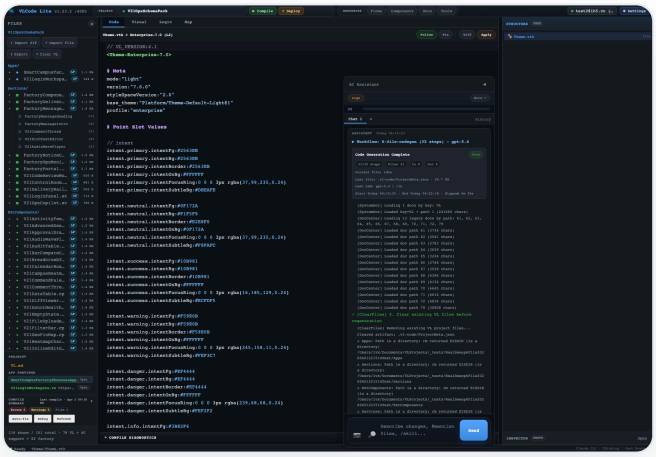
03 · AN AI-NATIVE LANGUAGE

A language designed for AI — decoupled on every axis.

VL is a programming language **designed for AI to generate** — not a human language the model is forced to imitate. Its reliability comes from decoupling: every concern is separated so it can be generated, verified, and swapped on its own.

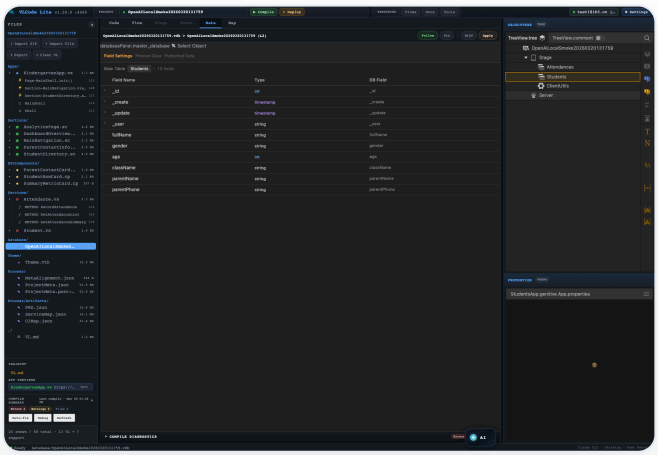
- **Front / back** — Sections + Components `.sc + .cp` vs Services `.vs`
- **Componentized architecture** — everything is a composable unit
- **Look / structure** — visuals and design tokens live in **THEME**, apart from logic
- **Build unit / deploy unit** — what you author is not what you ship
- **Data / data-usage** — schema `.vdb` vs **VirtualTable**

Decoupling is *why* AI generation is reliable here: smaller, independent surfaces mean less ambiguity going in and fewer hallucinations coming out.



THEME — visuals, decoupled

Design tokens and styling are a separate layer, so the same structure can be re-skinned without touching logic.



.vdb / VirtualTable — data, decoupled

Schema is defined once; how each screen consumes it is a separate, swappable concern.

04 · LANGUAGES, LEVELED

To AI, every language is about the same.

So we built the one that costs the AI the least.

AI didn't only flatten the distance between **natural** languages — English, 中文, 日本語 — it flattened **programming** languages too. To a model, Python, Java and JavaScript are much of a muchness; it reads and writes any of them. The old reason to choose a language — human familiarity — barely registers with the model.

But languages are *not* equal in what they cost the AI:

- **Tokens per I/O** — how much context every step burns just to say the same thing
- **How logic is expressed** — explicit and checkable, or implicit and buried
- **How an app is composed** — clean, swappable units, or tangled cross-file state

> **If the model is indifferent to the language, the right move is to design the one that costs it the least.**

So we did. VL is a language whose intended user is the **AI**, not the human — fewer tokens, explicit logic, composable by construction. (People still pick it up in an afternoon.)

05 · COMPONENTIZE EVERYTHING

Bank the token-compute. Share it to everyone.

Every asset is encapsulated as a reusable component. Think of it as **banking the expensive token-compute of AI**: generate once, store it, and share it to every developer. The cost of intelligence is paid a single time, then amortized across everyone — higher output, lower spend.

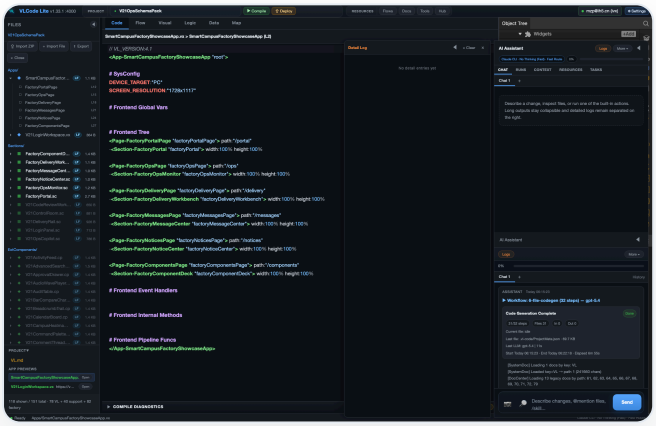
06 · OPEN & DECOUPLED

Nothing is a black box. Bring your own.

The whole development surface is composable and open. Every layer is decoupled and yours to upload:

DAGs / Workflows · AI Chat scopes · Components · Tools · Docs · Skills

The platform is a substrate; your assets ride on top. Upload your own workflows, wire your own tools, attach your own docs and skills — the system orchestrates them, it doesn't hide them.



AI Chat — a decoupled scope

Chat is one composable surface among many — not the whole product bolted to a prompt box.

07 · STORE, DON'T BURN

Don't buy more GPUs. Store the compute.

The Component Factory is a compute-storage center.

The industry is racing to add GPUs and power, as if intelligence were only ever something you *spend*. We think that's the wrong axis. Most of that compute is **storable**: once AI has paid the tokens to build something correct, the result can be banked — and reused forever, by everyone.

That is what our **Component Factory** is — not a parts bin, but a **compute-storage center**. Every component in it is intelligence that was generated once and never has to be generated again.

THE INDUSTRY

More GPUs · more power

Treats every app as fresh compute to be spent — so the bill, and the hunger for hardware, never stop growing.

VLC

Generate once · store · assemble

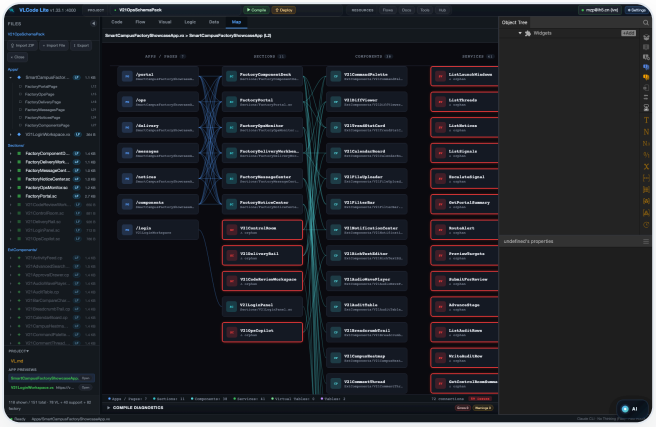
Pay the token cost a single time, bank the result, and assemble from the bank — the marginal compute trends toward zero.

> **Picture 10,000,000 components. At that point ~99.999% of every website, platform and enterprise service on Earth is already covered — and the compute needed to "write code" all but disappears.**

08 · EVERYTHING IS GRAPHICAL

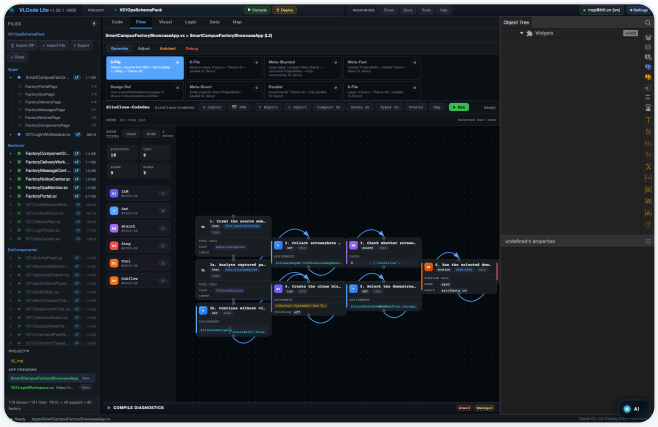
You reason in pictures, not buried text.

Every dimension of a VL application has a visual surface. You see the system, instead of reconstructing it in your head.



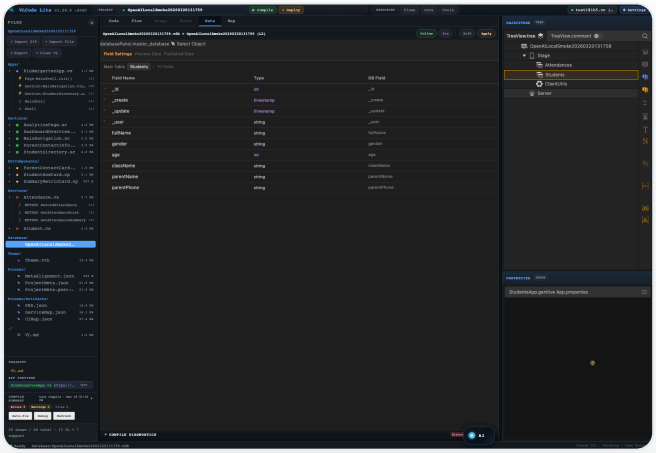
Architecture → Map

The whole application laid out as one navigable graph of sections, components and services.



Process → DAG + Event Panel

Generation, automation and logic run as visible nodes and edges — inspectable at every step.



Data → ER

Entities, fields and relationships shown as a model — not guessed from scattered code.

09 · BACKEND AS COMPONENTS

The backend is packaged too — billed by the cloud.

The backend itself is a set of pre-packaged components. Publish a VL application and it runs and **bills directly on cloud-compute usage** — no servers to wire, no ops to babysit. You ship product, not plumbing.

10 · ZERO LOCK-IN

Export the source. Deploy anywhere. 0 binding.

Your output is yours. Export the full source code and deploy it independently — **zero binding**. VLC earns your stay by being useful, never by trapping you. Use the cloud when it helps; walk away with your code whenever you want.

PUBLISH

Run on managed cloud

Backend components deploy and scale themselves; you pay for compute you actually use.

OR EXPORT

Self-host the source

Take the generated source out and run it on your own infrastructure. No platform tax.

WHY IT HOLDS TOGETHER

From "looks right" to verified.

+ VERIFIABLE, NOT HOPEFUL

Every node can be checked

Because each build is a DAG, every node can run lint, compile, preview, a test, or a human gate. Runs are reproducible, rerunnable and auditable — leaving a ledger of artifacts and evidence.

+ RIGHT MODEL PER NODE

Capability where it counts

Route expensive models to the hard nodes and cheap models or tools to the routine ones. You spend intelligence where it matters and save it everywhere else.

Raw LLM chat vs VLC

	Raw LLM chat	VLC — AI in a DAG
Next step	Unpredictable	**Defined by the graph**
Your role	Babysit every output	**Orchestrate the flow**
Reproducibility	One-off conversation	**Versioned, rerunnable DAG**
Error handling	"Try again"	**Checkpoint → retry → resume**
Trust	"Looks right"	**Verified by compile / test / gate**
Reuse	Copy-paste prompts	**Components, tools, skills**
Cost	Best model for everything	**Right model per node**
Lock-in	—	**Export source · 0 binding**

THE BET

AI's creativity, harnessed — not hoped for.

VL is the first deep domain of a local, visual agent runtime: a place where the model's brilliance at each point is composed into a line you can ship, verify, and own. Capability and creativity from the AI; stability and trust from the graph.

Tame AI inside a DAG.

Open source (MIT). Local-first. Clone & run, or grab the prebuilt Mac / Windows app.

github.com/VisualLogic-AI/VL-Code

VisualLogic.ai-VL — language & samples

editor.visuallogic.ai

VL-Code · VisualLogic — an AI-native platform that decouples front/back, look/structure, build/deploy, and data/usage, packages everything for reuse, renders it all as graphs, and lets you export the source with zero lock-in.