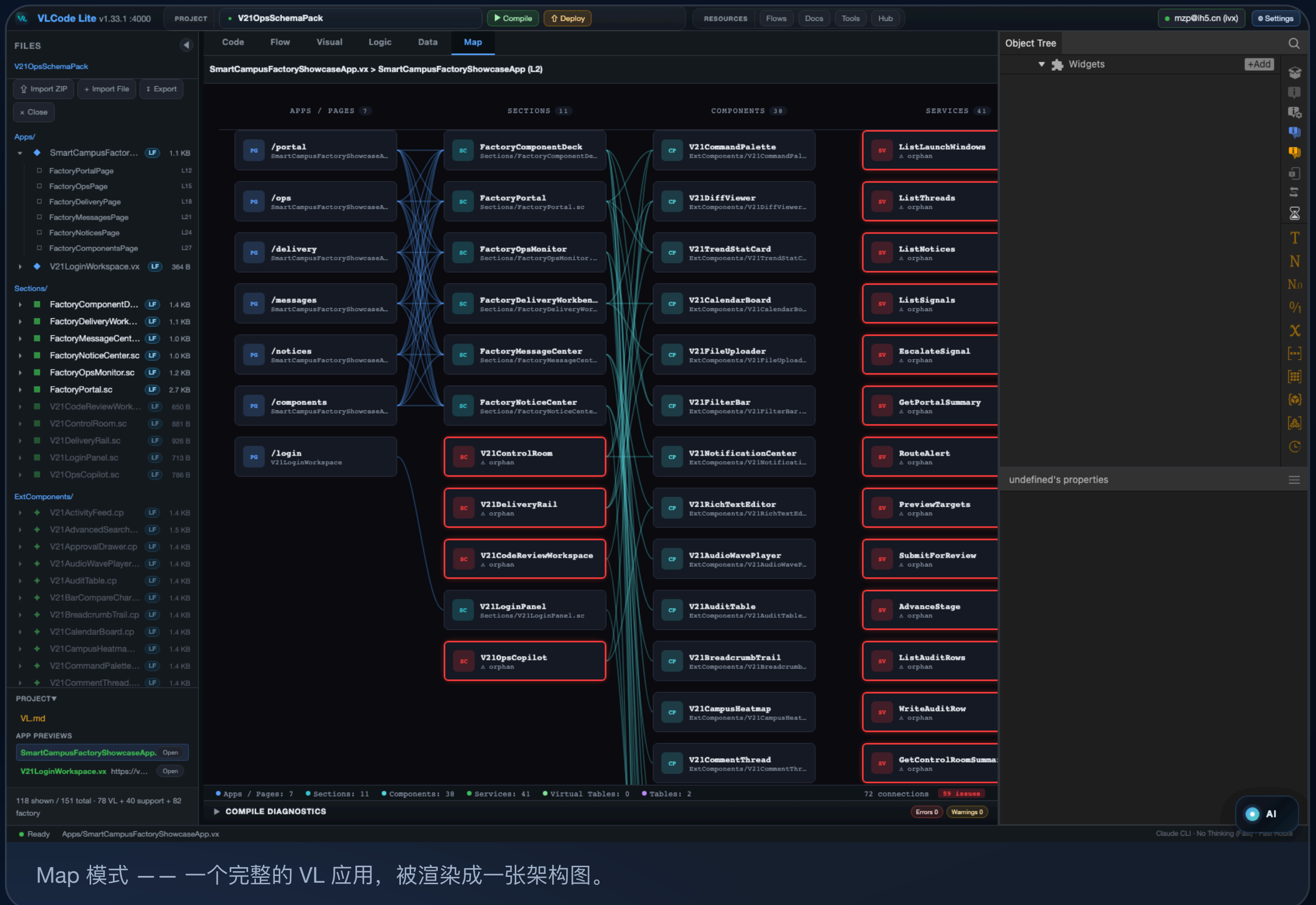
VisualLogic

AI 原生开发平台

把 AI
驯服进 DAG。

今天的 AI 能写出惊艳的**「点」**，VLC 把这些点连成一条**可信任的「线」**——既保住模型的能力与创造力，又用一张图把执行逻辑稳稳定住。



Map 模式 —— 一个完整的 VL 应用，被渲染成一张架构图。

VL-Code　开源（MIT）· 本地优先　github.com/VisualLogic-AI/VL-Code

—— 01 · 核心思想

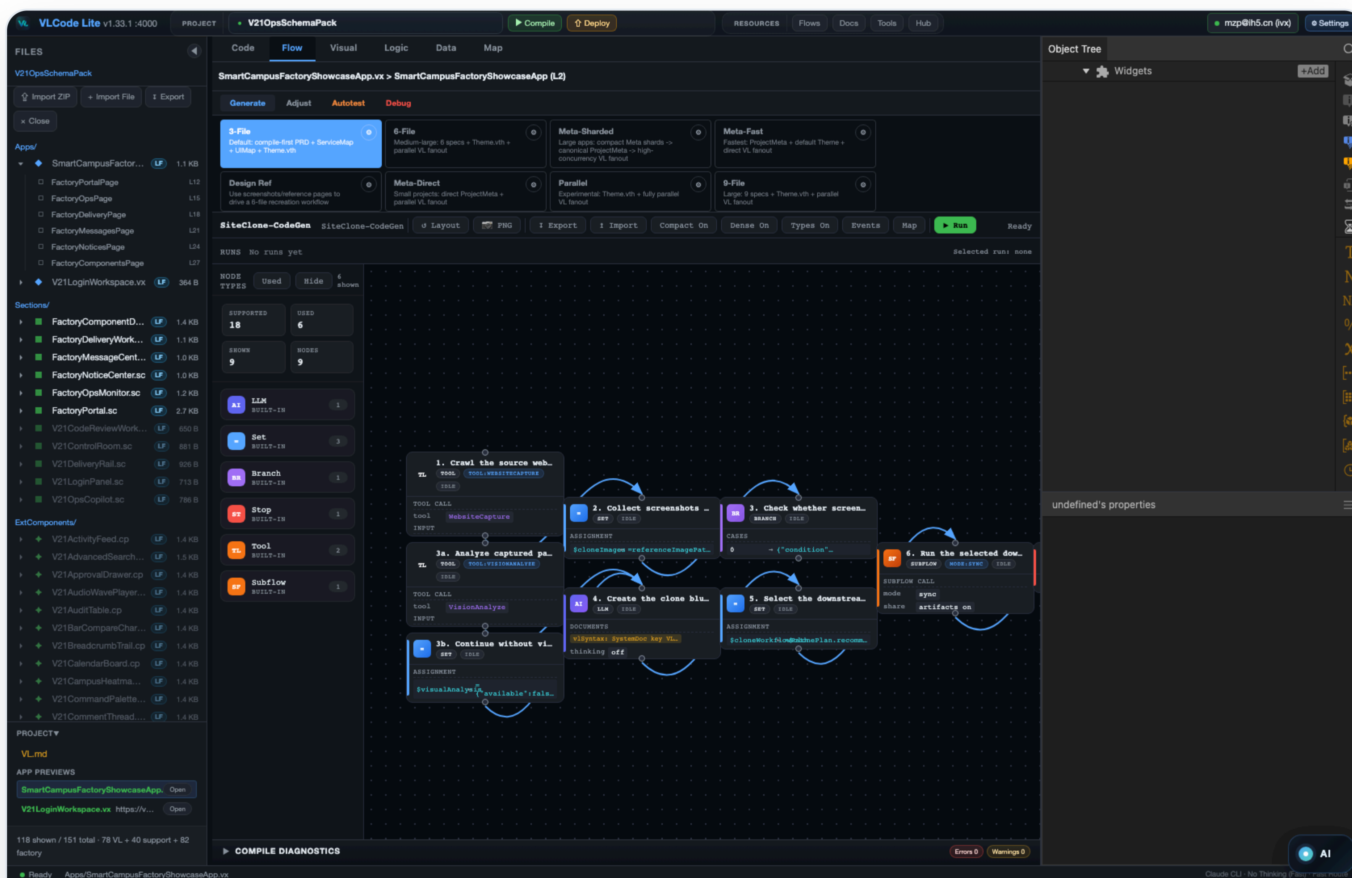
AI 擅长一个「点」，却守不住一条「线」。

所以 VLC 把 AI 放进了一张图里。

今天最强的编程 AI——Claude Code、Codex——已经足够惊艳。但它们的工作方式是**「点」**：一次输入、一次输出。作为开发者，你无法预估这次的输出，也就无法预估*下一步的起点*。于是你必须盯着每一步，还要在每一步给出有价值的回应。这对开发者要求极高，也意味着你并没有真正被解放。

> **真正的问题，不是「换个更聪明的模型」，而是：如何同时保住 AI 的能力与创造力，又能守住执行逻辑的稳定？**

VLC 的答案，是把 AI 放进一张 **DAG** 里。创造力发生在**每个节点内部**，确定性保存在**节点之间的连边**上。模型尽可以在每一步天马行空，而图保证它的下一步，正好落在你预期的位置。你不再是保姆，而是编排者。



Flow 模式 —— 每一次构建都是一张可验证的 DAG：生成 → 校验 → 修复 → 预览。

同一个 AI，人与人之间却有 十倍的差距。

把它拉平的，是一张 DAG。

在单个**「点」**上——一次输入、一次输出，再复杂也只是一个点——人已经追不上模型了，这场比赛结束了。但真实的应用从来不是一个点，而是由许多点、层层叠叠构成的一条**「线」**。今天真正的工作，是去*驾驭*这条线上的每一处输入与输出。

> **面对同一个 AI，有人能产出十倍的价值，有人却束手无策。差别不在模型，而在于你能不能守住那条线。**

这正是 **DAG** 所做的：它是驱动 AI 一个节点一个节点走完整条线的路径规划——而这张规划，AI 自己就能画出来。人与人之间的差距随之被抹平，因为扛起这条线的，是结构，而不是某一个人。

前提一

逻辑可拆解，也可重组

任何复杂过程都能干净地拆成点与边，它背后的算力也按同样的方式拆分。可被拆解的，就可被编排。

前提二

AI 极其擅长逻辑

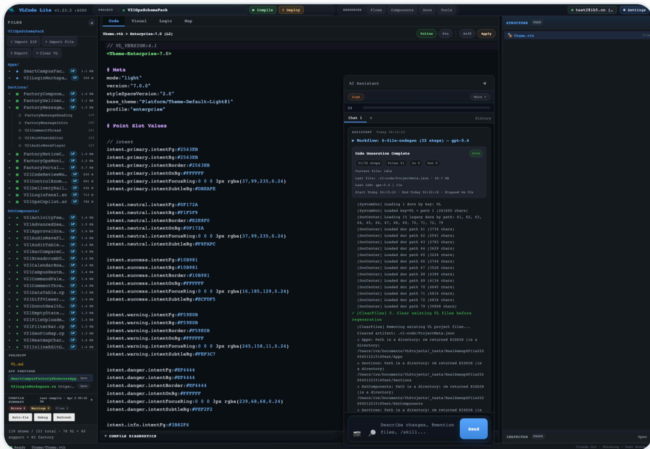
把显式的结构交给模型，它就能可靠地规划、填充、校验每一步——远比让它把整个程序都装在脑子里要稳得多。

一门为 AI 而设计的语言 —— 在每一个维度上解耦。

VL 是一门**为 AI 生成而设计**的编程语言，而不是让模型去模仿人类语言。它的可靠，来自彻底的解耦：每一个关注点都被拆开，可以被独立生成、独立验证、独立替换。

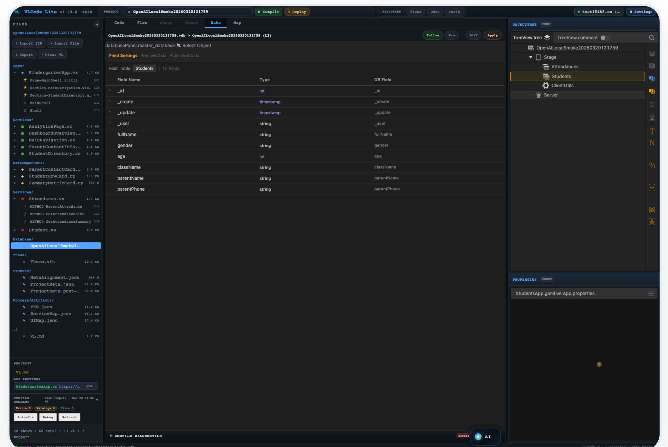
- **前后台解耦** —— Section + Component `.sc + .cp` 与 Service `.vs`
- **组件化架构** —— 一切皆可组合单元
- **视觉与结构解耦** —— 样式与设计令牌放在 **THEME**，独立于逻辑
- **开发单元与部署单元解耦** —— 你写的，不等于你发的
- **数据与数据使用解耦** —— Schema `.vdb` 与 **VirtualTable**

解耦正是 AI 生成可靠的根本原因：面更小、更独立，输入的歧义更少，输出的幻觉也更少。



THEME —— 视觉解耦

设计令牌与样式是独立的一层，同一套结构可以换肤，而不必动逻辑。



.vdb / VirtualTable —— 数据解耦

Schema 只定义一次；每个页面如何使用数据，是另一件可替换的事。

在 AI 眼里，语言都差不多。

于是我们造了一门让 AI 最省力的语言。

AI 不只抹平了**自然语言**之间的距离——英语、中文、日本語——它同样抹平了**编程语言**。在模型看来，Python、Java、JavaScript 其实大同小异，哪一种它都能读能写。过去挑语言的理由——人是否熟悉——对模型几乎不再重要。

但语言对 AI 的*成本*，并不相等：

- **每次 I/O 的 token** —— 说同一件事，每一步要烧掉多少上下文
- **逻辑如何表达** —— 是显式、可校验，还是隐式、被埋起来
- **应用如何拼装** —— 是干净、可替换的单元，还是纠缠的跨文件状态

> **既然模型对语言无所谓，那么正确的做法，就是设计一门让它最省力的语言。**

我们正是这么做的。VL 这门语言，预设的使用者是 **AI**，而不是人——更少 token、逻辑显式、天生可组合。（而人，学起来一个下午也就会了。）

05 · 一切组件化

把 token 算力存起来，共享给每个人。

每一个资产都被封装成可重用的组件。可以把它理解为**把 AI 昂贵的 token 算力存起来**：生成一次、存下来、共享给每一个开发者。智能的成本只付一次，却被所有人摊薄——产出更高，花费更低。

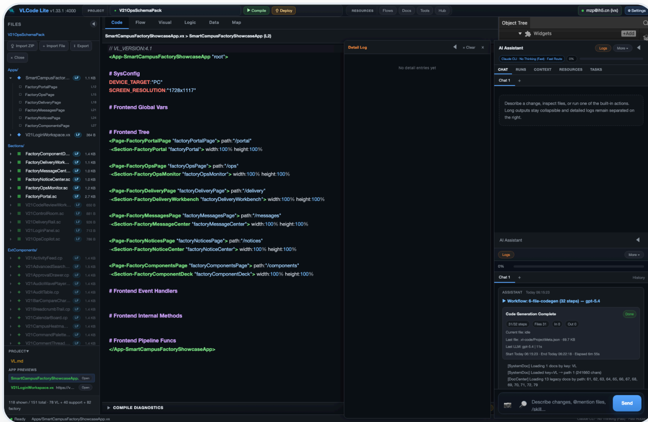
06 · 开放且解耦

没有黑箱，一切自带。

整个开发面都是可组合、可开放的。每一层都解耦，且可以由你自行上传：

DAG / 工作流 | AI Chat 作用域 | 组件 Components | 工具 Tools | 文档 Docs | 技能 Skills

平台是底座，你的资产跑在上面。上传自己的工作流、接上自己的工具、挂上自己的文档与技能——系统只负责编排它们，而不会把它们藏起来。



AI Chat —— 一个解耦的作用域

对话只是众多可组合面之一，而不是「把整个产品绑在一个输入框上」。

别再买显卡，把算力存起来。

组件工厂，就是一个算力存储中心。

整个行业都在拼命堆显卡、抢电力，仿佛智能只能是被*花掉*的东西。我们认为这条轴选错了。其中大部分算力，其实是**可以存下来**的：AI 一旦付出 token 把一件事做对，这个结果就能被存起来——并被所有人永久复用。

这正是我们的**组件工厂**——它不是一个零件箱，而是一个**算力存储中心**。里面的每一个组件，都是只被生成过一次、此后再也不必重新生成的智能。

行业

堆更多显卡 · 抢更多电力

把每个应用都当成要重新花掉的算力——于是账单和对硬件的渴求，永远停不下来。

VLC

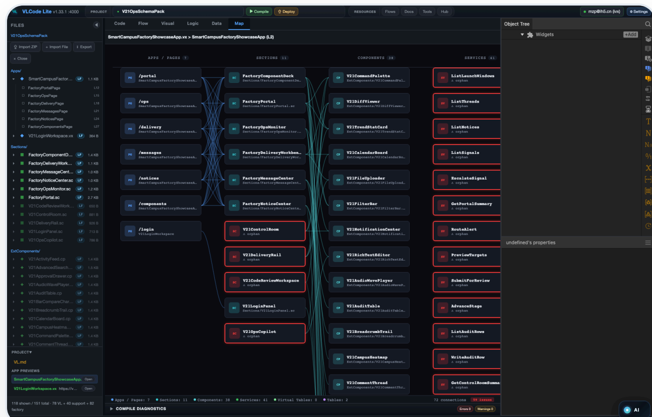
生成一次 · 存下 · 拼装

token 成本只付一次，把结果存进库，再从库里拼装——边际算力趋近于零。

> **设想有一千万个组件。届时全球约 99.999% 的网站、平台与企业服务都已被覆盖——而「写代码」所需的新算力，几乎归零。**

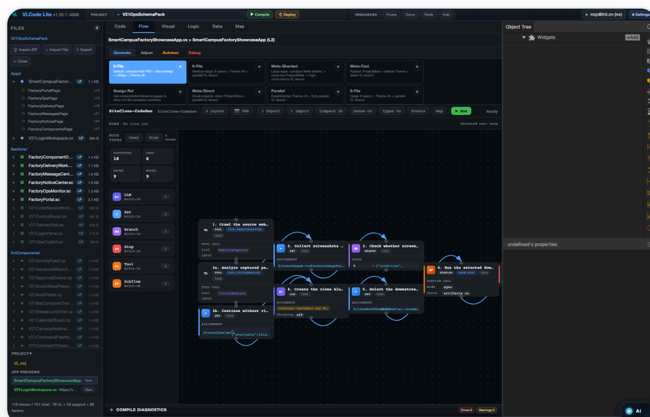
你用「图」来思考，而不是埋在文字里。

VL 应用的每一个维度，都有一张对应的可视化面。你「看见」系统，而不必在脑子里把它重新拼出来。



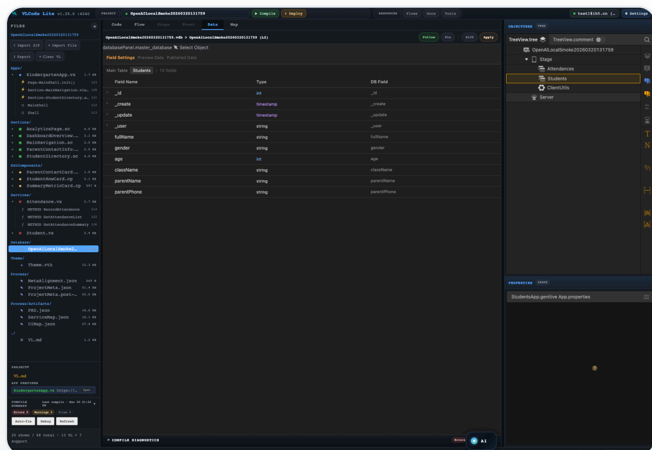
架构 → Map

整个应用铺成一张可导航的图：Section、Component 与 Service 一览无余。



过程 → DAG ＋ Event Panel

生成、自动化与逻辑，都是看得见的节点与连边，每一步都可检视。



数据 → ER

实体、字段与关系，以模型的方式呈现——而不是从散落的代码里去猜。

—— 09 · 后台即组件

后台也是封装好的组件 —— 按云计算收费。

后台本身，就是一组预先封装好的组件。发布一个 VL 应用，它就直接运行，并**按云计算用量收费**——不必接服务器，不必养运维。你交付的是产品，而不是管线。

—— 10 · 0 绑定

可导出源码，独立部署，0 绑定。

产出归你所有。完整源代码可导出、可独立部署——**0 绑定**。VLC 靠「有用」留住你，而绝不靠「锁死」绑住你。云有用就用云，想带着代码走，随时都可以。

发布

跑在托管云上

后台组件自动部署、自动伸缩，你只为真正用掉的算力付费。

或导出

自托管源码

把生成的源代码带走，跑在你自己的基础设施上，没有平台税。

—— 为什么它立得住

从「看起来对」，到「已被验证」。

＋ 可验证，而非靠运气

每个节点都能被校验

因为每一次构建都是 DAG，每个节点都可以跑 lint、编译、预览、测试，或一个人工关卡。运行可复现、可重跑、可审计，并留下工件与证据的账本。

＋ 每个节点用对模型

能力用在刀刃上

难节点交给昂贵的模型，常规节点交给便宜的模型或工具。智能花在该花的地方，其余处处省下来。

原始 LLM 对话 vs VLC

	原始 LLM 对话	VLC —— AI in a DAG
下一步	无法预估	**由图定义**
你的角色	盯着每个输出	**编排整张流程**
可复现	一次性对话	**版本化、可重跑的 DAG**
出错处理	「再试一次」	**断点 → 重试 → 续跑**
可信	「看起来对」	**编译 / 测试 / 关卡 验证**
重用	复制粘贴 prompt	**组件、工具、技能**
成本	什么都用最贵的模型	**每个节点用对模型**
锁定	——	**导出源码 · 0 绑定**

我们押的注

AI 的创造力，是被「驾驭」，而不是被「祈祷」。

VL 是一个本地、可视化 Agent 运行时的第一个纵深领域：在这里，模型在每个点上的才华，被编排成一条你能交付、能验证、也真正拥有的线。能力与创造力，来自 AI；稳定与信任，来自那张图。

把 AI 驯服进 DAG。

开源（MIT）· 本地优先。可以 clone 自己跑，也有 Mac / Windows 预编译版直接下。

github.com/VisualLogic-AI/VL-Code

VisualLogic.ai-VL —— 语言与示例

editor.visuallogic.ai

VL-Code · VisualLogic —— 一个 AI 原生平台：前后台、视觉与结构、开发与部署、数据与使用，全部解耦；一切封装以重用；一切渲染成图；并且可以导出源码，0 绑定。