AI-Powered Internal Tool Builder

Vision, Architecture & Value Proposition

The Goal

> *"Enable non-technical employees to build functional internal tools using natural language, without writing code."*

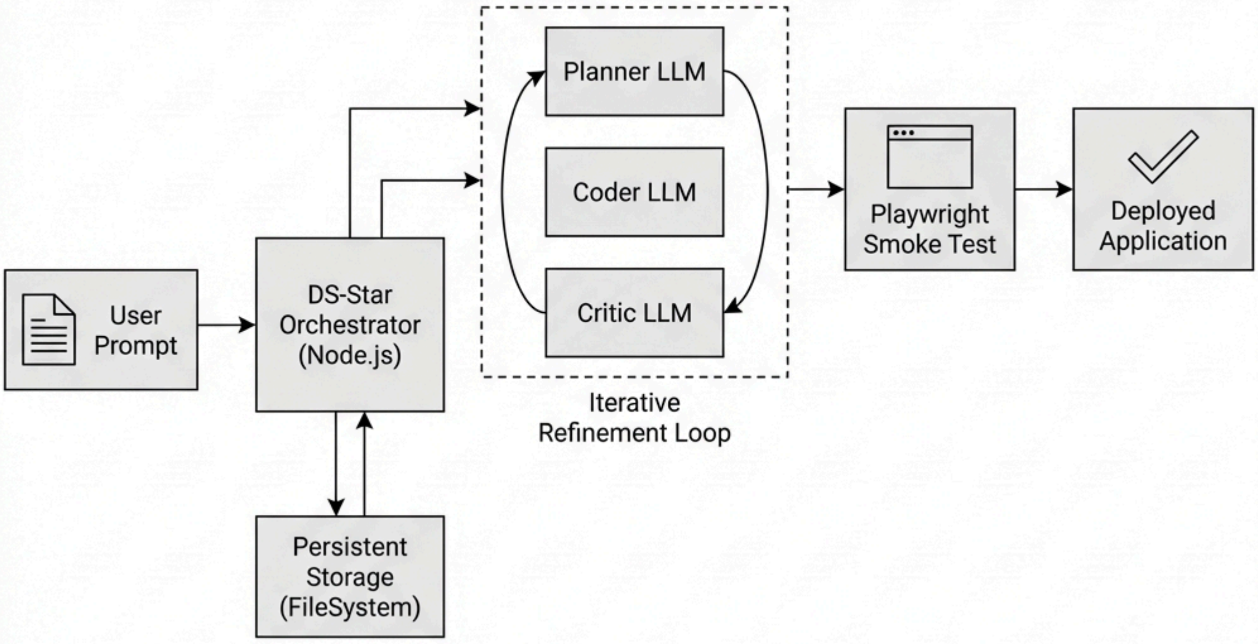
The Challenge

Traditional app development requires:

- **Technical expertise** (HTML, CSS, JavaScript)
- **Development cycles** (days to weeks)
- **IT involvement** for every small tool

Result: Simple internal tools (calculators, dashboards, trackers) either don't get built, or take weeks to deliver.

The Vision: From Prompt to Product


User Prompt
DS-Star Orchestrator (Node.js)
Persistent Storage (FileSystem)
Planner LLM
Coder LLM
Critic LLM
Iterative Refinement Loop
Playwright Smoke Test
Deployed Application

A user types: *"Create an inventory tracker for my department"*

Within minutes, they receive a **working, tested, deployable application**.

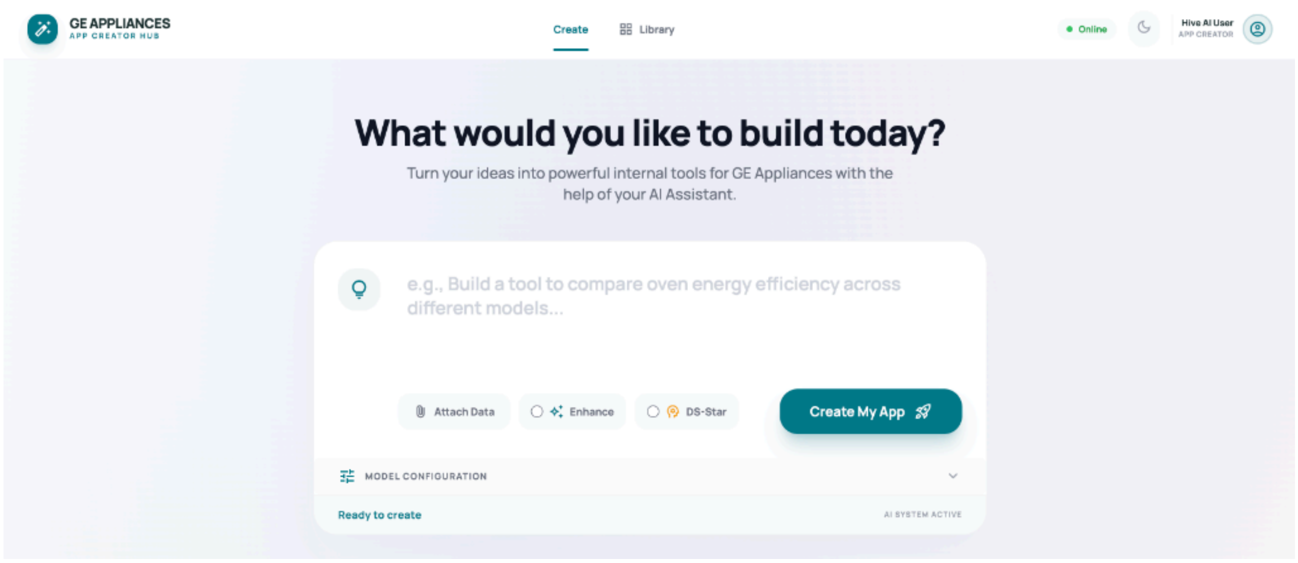
Live Demo

🚀 **Try it now**: https://geappliances-ai-tool-prototype.onrender.com

---

The User Interface

Home: Describe What You Want to Build

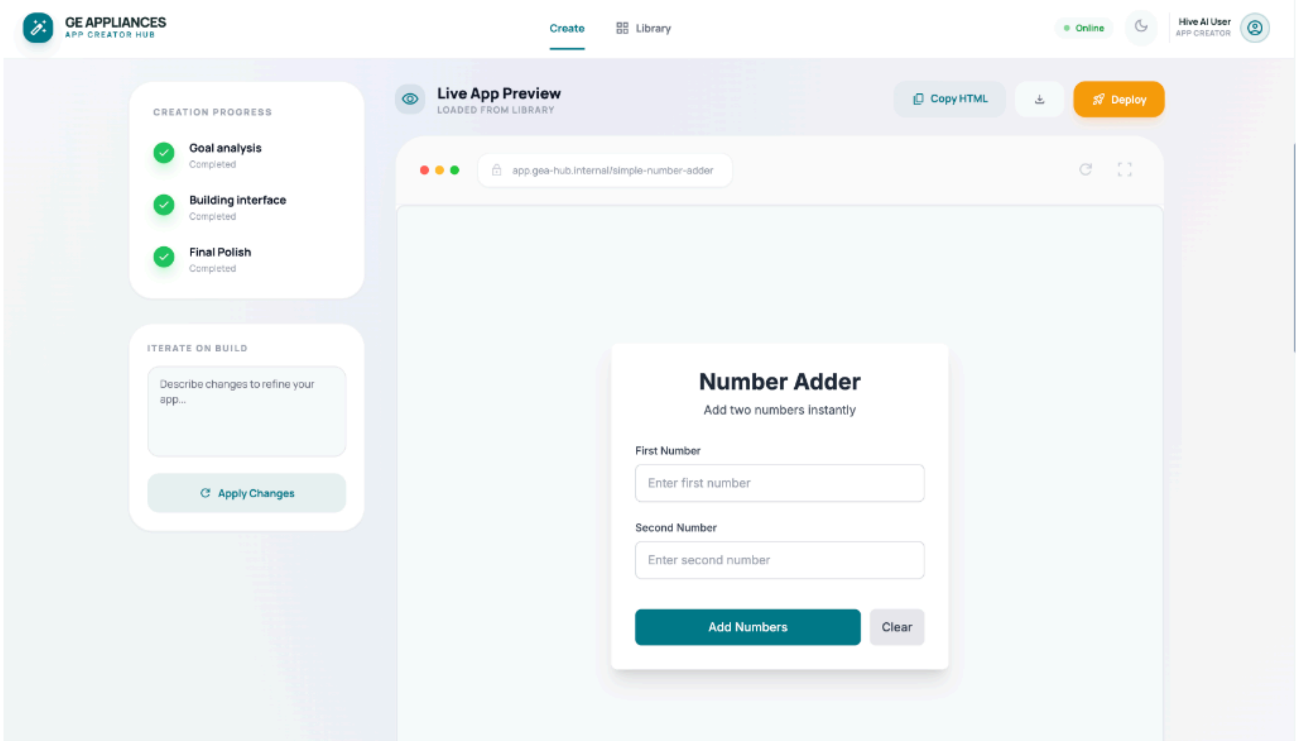


What would you like to build today?

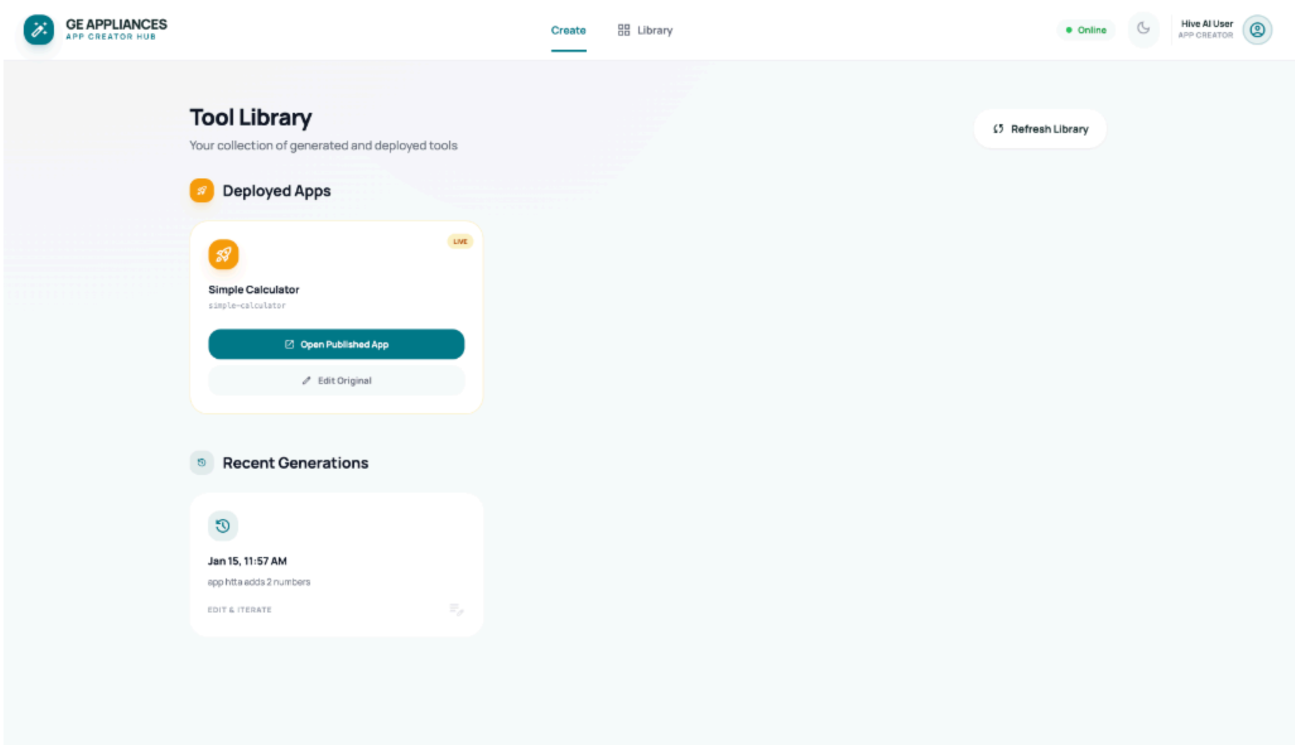
Turn your ideas into powerful internal tools for GE Appliances with the help of your AI Assistant.

e.g., Build a tool to compare oven energy efficiency across different models...

Live Preview: Watch Your App Come to Life



Library: Manage Your Deployed Tools


GE APPLIANCES
APP CREATOR HUB
Create
Library
Online
Hive AI User
APP CREATOR
Refresh Library

Tool Library

Your collection of generated and deployed tools

Deployed Apps

LIVE
Simple Calculator
simple-calculator
Open Published App
Edit Original

Recent Generations

Jan 15, 11:57 AM
app htta adds 2 numbers
EDIT & ITERATE

Core Architecture Principles

1. Multi-Agent Collaboration

Three specialized LLM agents work together:

Agent	Role
Planner	Translates user intent into structured requirements
Coder	Generates complete, functional HTML/CSS/JS
Critic	Reviews output for security, completeness, and quality

2. Iterative Self-Correction

Unlike single-shot generation:

- Code is **tested automatically** using browser automation
- Failures produce **specific error messages** fed back to the LLM
- System iterates until the code **actually works**

3. Secure by Design

- Generated apps run in a **sandboxed environment**
- No access to external APIs or databases
- All AI calls and data storage go through **controlled runtime helpers**

What Makes This Different?

Traditional Code Gen	DS-Star Pipeline
Generates code, hopes it works	Generates, tests, fixes until it works

Single LLM call	Multiple specialized agents
No validation	Playwright-based smoke tests
Static output	Self-correcting loop

Expected Outcomes

For Business Users

- **Hours → Minutes**: Build tools in the time it takes to describe them
- **No Dependencies**: Don't wait for IT to build simple dashboards
- **Iterate Quickly**: Refine with natural language

For IT/Engineering

- **Reduced Ticket Volume**: Fewer requests for basic internal tools
- **Guardrails Built-In**: Security policies enforced automatically
- **Audit Trail**: Every generation logged with full history

For the Organization

- **Democratized Development**: Anyone can create tools
- **Standardized Output**: Consistent, branded, secure applications
- **Knowledge Capture**: Tools can be deployed and shared

Use Cases

1. **Operations**: Inventory trackers, scheduling tools, status dashboards
2. **Finance**: Calculators, report generators, data visualizers
3. **HR**: Survey tools, onboarding checklists, training trackers
4. **Engineering**: Data explorers, log analyzers, internal utilities

---

Technical Differentiators

```
┌─────────────────────────────────────────────────────────┐
│                 ITERATIVE REFINEMENT                    │
│                                                         │
│   Prompt → Plan → Code → Critique → Test → Deploy      │
│               ↑                         │               │
│               └────── Fix & Retry ←─────┘               │
│                                                         │
└─────────────────────────────────────────────────────────┘
```

- **Accumulated Error Memory**: Past failures inform future iterations
- **Structured Error Feedback**: LLM receives actionable fix instructions
- **Runtime Helpers**: Built-in AI and storage APIs for generated apps

---

Challenges & Considerations

Security Risks

- **Prompt Injection**: Malicious prompts could attempt to generate harmful code
- **Data Exposure**: Generated apps must not access sensitive internal systems without proper auth
- **Code Execution**: Even sandboxed apps carry risk if sandbox is bypassed

Mitigations: CSP headers, forbidden API blocking, runtime proxies, code scanning

Reliability Concerns

- **LLM Hallucinations**: Generated code may appear correct but contain subtle bugs
- **Non-Determinism**: Same prompt can produce different results across runs
- **Complex Requirements**: Multi-step workflows may exceed single-page app capabilities

Mitigations: Automated smoke testing, iterative correction, human review before deployment

Current Limitations

- **No Database Access**: Apps can only use local storage via runtime helpers
- **Single-Page Only**: Cannot generate multi-file or backend applications
- **Styling Constraints**: Output quality depends on LLM's understanding of design

Honest Assessment

Aspect	Current State
Simple tools (calculators, forms)	✅ Works well
Data visualization	⚠️ Requires iteration
Complex workflows	❌ May need manual intervention
Production-ready apps	⚠️ Needs human review

Summary

Aspect	Value
Speed	Minutes, not weeks
Quality	Tested and validated before delivery
Security	Sandboxed, no external access
Scalability	Self-serve for all employees
Flexibility	Iterate with natural language

Built with the DS-Star Iterative Pipeline