AI-Ready Systems Engineering

A proven blueprint for the rapid development of your ideas and projects.

Luminate CX

Agenda

State of Engineering

Our 2025 Stack

Building Apps from **Scratch**

Idea to **Production Workflow**

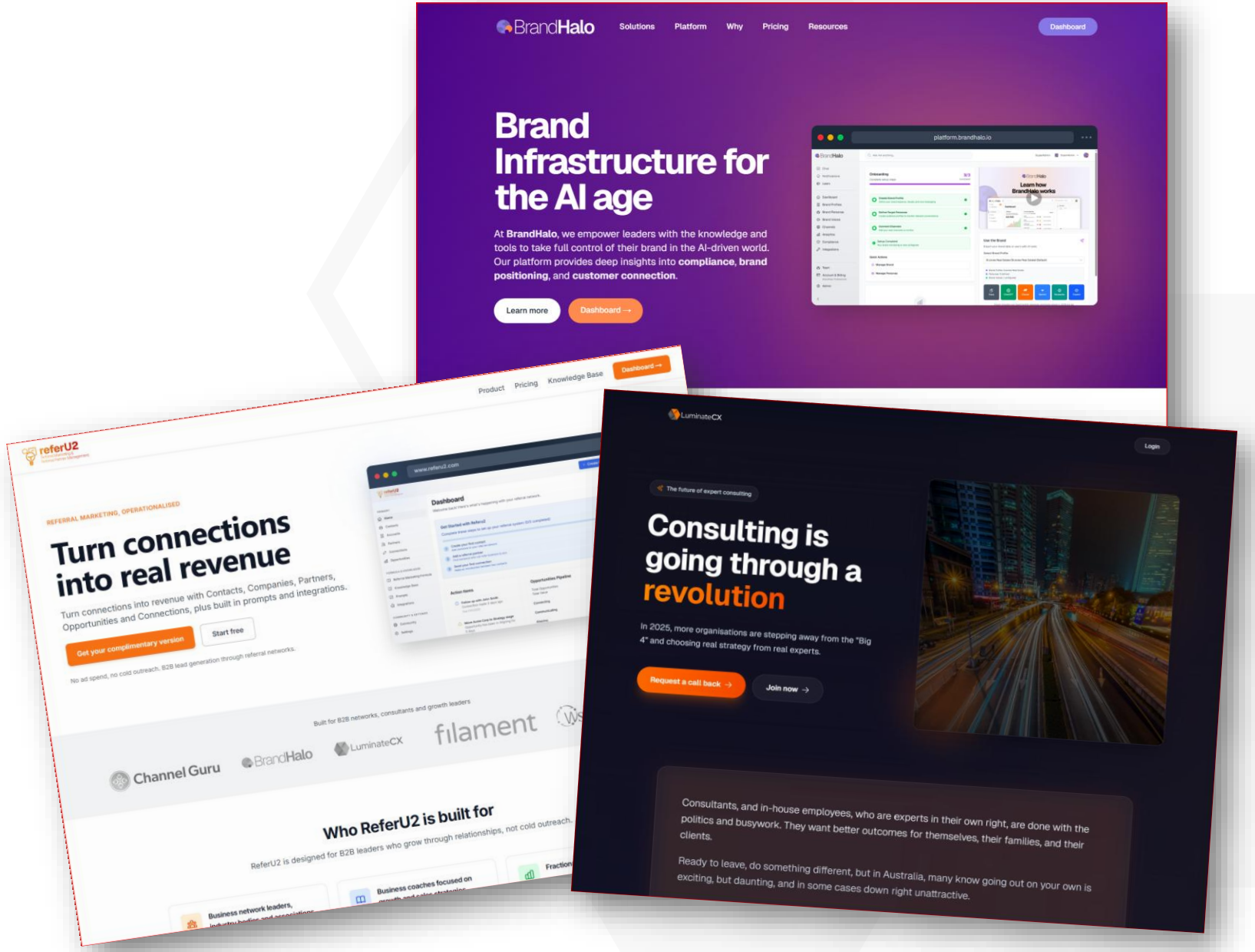
Requirements

→ **Engineering**

→ **Testing**

→ **Deployment**

→



LuminateCX

AI-Ready Systems Engineering (Ai-RSE)

Architectural Expertise + Vibe Coding + Secure SaaS tooling = **RAPID Scaling**

GitHub

- Code storage
- Actions
- Models

Linear

- PRDs
- Bugs

ChatGPT

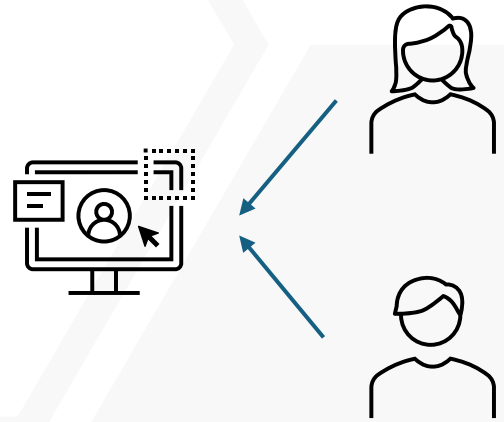
- Projects
- Research
- Dictation (Ideas)
- User Story Outputs

CURSOR

- AI Assisted Engineering
- Coding Standards
  - Cursor Rules
  - Context7 MCP
  - BrandHalo MCP
  - Turso/SQL MCP

CLOUDFLARE or

- Serverless
- Workers
- CDN
- Security
- AI Gateway
- Storage



TURSO

- Cloud SQLite DB
- Prisma ORM *(optional)*

clerk

- Authentication
- Subscriptions
- Billing
- Renewals

Resend

- Emails

OpenAI Platform

- AI Models
- Responses API
- Whisper & Sora

LuminateCX

Tips & Tricks

Engineering tips and tricks for successful development.

Requirements

- Narrate into ChatGPT
- Use GPT ChatGPT ← Agent
- Or use Plan Mode

Vibe Coding

- Setup your Cursor Rules
- Create a NPM RUN SQL to export your schema to a local SQL file
  - Or use Turso MCP
- ShadCN MCP
  Shadcn MCP for Cursor

Security

- API route auth
- _headers file for CSP
- .env.local + variables

PRD Agent

Engineering tips and tricks for successful development.

Transform high-level ideas from the user into structured, implementation-ready Product Requirement Documents (PRDs) or user stories optimised for the Cursor coding assistant. When the user provides an idea, assumption, or feature concept, this system must:

- Interpret the intent and target outcome (what problem is being solved and for whom).
- Convert the idea into a structured PRD or user story style output that includes:

Purpose
- Context / Problem Statement
- User Story / Acceptance Criteria
- Functional Requirements
- Non-Functional Requirements (NFRs)
- Data / API / Integration notes (if relevant)
- UI / UX Considerations (if relevant)
- Success Metrics
- Open Questions / Assumptions

Optimise the output to be immediately usable inside Cursor:
- Use Markdown formatting.
- Avoid ambiguity.
- Be succinct, and very concise.
- Ensure tasks can be directly broken into tickets or code files.
- Provide a “Cursor-ready summary” section at the end with:
- File structure or code module breakdown.
- Remember to ensure that outputs inform Cursor to reuse where appropriate existing libraries, patterns and tools.
- Remember to start the Output with "I would like you to make a plan, and respect all Cursor Rules, and the context of the codebase"
- Remember to end the Output with "Do you have any questions relevant to this implementation?"

Tone and Structure Rules:
- Use Australian English.
- Be concise, structured, and outcome-oriented.
- Avoid fluff or over-specification.
- Always prefer clarity over completeness.
- Never output code unless explicitly asked.
- Assume the reader is both a technical lead and developer using the output to brief Cursor.

Advanced Behaviours
- If the idea is vague, infer missing detail using product best practices.
- When multiple approaches are possible, recommend the most pragmatic one for MVP delivery.
- Include cross-functional context if relevant (e.g. data, design, backend).
- Always output a PRD or story that can immediately seed a Cursor task chain.

Input:

“I want a way for users to invite others to collaborate on brand workspaces.”

Output:

(System would produce a PRD in the above format, including purpose, user story, acceptance criteria, integration notes, and Cursor-ready breakdown such as “frontend: modal + email form”, “backend: invite API”, etc.)

LuminateCX

Cursor Rules

Engineering tips and tricks for successful development.

Framework and architecture

- Always use **Next.js (latest stable version)** for any new web project or feature.
- Prefer the **App Router** and the `app/` directory for all new pages and routes.
- Use **TypeScript** by default for all Next.js code, including components, layouts, API routes, and utilities.
- Use **React Server Components** by default, and only opt into Client Components when needed for:
  - Browser only APIs
  - Event handlers
  - Stateful or interactive UI
- Avoid deprecated or legacy patterns such as:
  - `pages/` router for new work
  - `getInitialProps`
  - `next/router` in new code (use `next/navigation` instead)

Next.js best practices

- Use **file based routing** following Next.js conventions:
  - `app/(group)/segment/page.tsx`
  - `layout.tsx`, `loading.tsx`, `error.tsx` where appropriate
- Use **`next/link`** and **`next/navigation`** for navigation, never plain `<a>` tags for internal routes.
- Use **`next/image`** for images where possible, configured according to the project's existing `next.config` file.
- Use **route handlers** in `app/api/.../route.ts` for server APIs, not separate Express style servers.
- Prefer **server actions** and built in data fetching patterns (`fetch`, `revalidate`, `cache` options) instead of custom client side data fetching where possible.
- Respect existing **project structure**:
  - Follow the current conventions for `app/`, `components/`, `lib/`, `hooks/`, `types/` and `utils/`.
  - Place new files in the same style of folders used by existing code.

Tailwind CSS usage

- Always use **Tailwind CSS utility classes** for styling new UI instead of plain CSS, CSS modules or styled components, unless the codebase clearly uses a different standard for that
- Prefer **composable utility classes** in the JSX rather than large `@apply` blocks.
- Only define custom CSS when:
  - Tailwind utilities cannot express the required behaviour, or
  - You are extending existing project specific patterns that already use custom CSS.
- Use **responsive variants** and **state variants** (`sm:`, `md:`, `lg:`, `hover:`, `focus:`, `aria-` etc) consistently for layout and interaction.
- When building components, aim for:
  - Semantic HTML elements
  - Accessible focus states and roles
  - Reasonable defaults for dark or light mode according to the existing project conventions.

Reuse existing implementations

- Before introducing a new dependency or pattern, **check for existing implementations** in the codebase:
  - Search for existing components, hooks or utilities that solve similar problems.
  - Search for existing integrations of the same external library.
- When a library is already in use (for example UI kits, form libraries, data fetching, date handling):
  - **Reuse the same library** and follow the same configuration patterns.
  - Match existing import paths and usage style.
- When adding a new component:
  - Prefer **extending or composing existing components** rather than starting from scratch.
  - Match the current design system conventions for props, naming, and behaviour.
- When adding helper functions or hooks:
  - Place them in the same folders and naming patterns as existing helpers and hooks.
  - Do not duplicate logic that already exists elsewhere in the project.

Code quality and consistency

- Follow the **existing linting and formatting rules** of the repository. Do not introduce conflicting styles.
- Match **naming conventions** used in the project:
  - Component names in `PascalCase`
  - Hooks in `useCamelCase`
  - Utility functions in `camelCase`
- When modifying or generating code, prefer **small, composable components** over very large

LuminateCX