Computer Science Department

Software Engineering

Bachelor's Thesis

DiagraVinci

Flexible Technical Diagramming Through Unified Visual, Code and AI Editing

Levko Beniakh

Supervisor
KSE, Vadym Yaremenko, PhD, vyaremenko@kse.org.ua

Submission date
16 June 2026

Academic Integrity Statement

I, undersigned, hereby declare that this capstone project is the result of my own work.

- All ideas, data, figures and text from other authors have been clearly cited and listed in the bibliography.
- No part of this project has been submitted previously for academic credit in this or any other institution.
- All code, diagrams, and third-party materials are either my original work or are used with permission and properly referenced.
- I have not engaged in plagiarism or any form of academic dishonesty.
- Any assistance received (e.g. from peers, tutors, or online forums) is acknowledged in the acknowledgements section.

I understand that failure to comply with these declarations constitutes academic misconduct and may lead to disciplinary action.

Place, date Kyiv, 16.06.2026

Signature 

Contents

Acknowledgements

First and foremost, I would like to express my great appreciation to all the teachers at KSE — the interest with which you were introducing us to the fields of your studies was entirely new to me. Coming from a public school where teachers seemed to be bored by their own subjects, I could now witness a true love for these topics, and it reshaped what I thought about much of the material I had learned. Your genuine interest and efforts in teaching and guiding us are by themselves enough reason to choose KSE as a place to pursue a bachelor's degree.

I would also like to express my thanks to all my classmates — I'm happy that I had the pleasure of receiving my degree alongside you, as you are amazing beyond any doubt. I cannot omit a special note of admiration for Marko, whose casual authenticity and positivity set an inspiring example.

Special thanks to the academic director and teacher, Artem Korotenko, for supporting and developing the programme, for teaching, and for his help throughout the four years of study.

Thank you to my supervisor and teacher, Vadym Yaremenko, for teaching, believing, and supporting. Since the classes when you first introduced diagramming for developers, and all the way through to the moment I now defend my final project on diagramming — you've been there, and helped me greatly.

Special thanks to everyone who tested the application — this capstone wouldn't be the same without you.

And lastly, thank you to all those I forgot to mention — if you think I should have, I would most certainly agree.

Abstract

When developing diagrams for modelling any system, we face a fundamental problem. Whilst there are countless tools for creating diagrams, none of them are actually useful across the entire span of the process. The problem is not just that each tool is more useful to specific people, or at specific stages of development, but even more that they all cease to provide the same value as the systems grow in complexity and scale. Elements cannot be filtered, folded, rearranged based on the person and the context of their access and use, diagrams cannot be navigated easily, or interactively tested to more closely observe the *behaviour* of the system rather than its look. These tools do not provide the possibility to focus on the right amount of context, abstract from the details, or conversely zoom into them as needed. All of these tools treat diagrams as static images of the system — without scaling or flexibility based on their application.

DiagraVinci is a browser-based tool that sets out to manage and control complexity and provide flexibility. It achieves this by: real-time synchronisation between text-based, visual, and AI editors (allowing the user to work on the diagram in whatever way is most convenient at the moment); a boolean selector system with filtering, selection, coloring, grouped actions, and session management (allowing users to create named "views" and switch between them); multiple layout algorithms (allowing different views of the same model depending on context); and an execution engine (allowing users to observe element interactions and test the system interactively).

At the core lies **`DiagramModel`**, with all visual data separately stored in **`ViewState`**. All 3 input types go through the same pipeline, and all merging and filtering is executed over the model without changing it. In other words, adding another view does not require changing the core model. Although the idea, and the prototype of the project were born more than a year ago, the actual work lasted for around 4 months, with 4-layered clean architecture being the primary design decision. The app is a fully-offline client application, deployed on GitHub Pages, with git workflows and GitHub Actions for all pipelines, and a supplementary library **`c7one`**, published as a GitHub Package.

While I have gone to great lengths and implemented all the crucial functionality, this project still has a long way ahead of it, with many features in mind, refinements and fixes yet to come, changes based on the user feedback, and more. I truly hope that not only will this project continue to live, but that it will thrive and overgo lengths further than before, eventually claiming the status of one of the most popular diagramming tools.

Keywords:

diagramming, *complexity management*, *DSL*, *AI-assisted*, *semantic model*, *React*, *Konva.js*

1 Introduction

1.1 The Problem with Diagramming Today

The moment a diagram first appears is usually when the developer already has an idea of the system to model. And before getting into the specifics of what to diagram, the first question is always which tool to use.

For a quick sketch, especially under time pressure, ChatGPT or Eraser.io is the natural choice. For presenting to a diverse or non-technical audience, Miro or Lucidchart is more appropriate. For a diagram that should live with code, be version-controlled, or be defined textually, Mermaid or PlantUML are preferred. And while this might reflect individual preference, it far more likely depends on the context — stage of the project, complexity, scale, changeability, and more. And the context changes.

A survey, held in the early stages of the development of DiagraVinci[1], showed that when respondents were asked as to which interaction mode they would prefer if a tool supported all three, most common response was "combination, depending on the task". Common pattern was "AI for the initial structure, then code for refinement, then visual to fix the layout before presenting." One product manager said their preferred approach was text-based approach. A tech lead said that "If the diagram is simple, I'd use AI and edit it. If complex, drag-and-drop is easier than explaining everything."

In other words, what the survey and personal feedback sessions discovered was that external factors play an even greater role than personal preferences, regardless of whether it is the phase of the project, its scale, or the audience. The problem is that changing the working format currently means switching tools entirely, or accepting results that fall short. Changing applications is a tedious process that requires recreating the diagram element by element. Sticking to the same tool, on the other hand, may not deliver the needed results, or may require workarounds and other cumbersome steps.

Scale and complexity magnify the problem significantly, introducing new concerns. Apart from the aforementioned issues, the existing tools become hardly manageable for complex systems as they lack features of filtering, folding, switching between different layouts, different views, selector-based coloring, navigable relationships, etc. Current tools are relatively rigid and unscalable.

1.2 Motivation

The core idea of DiagraVinci is quite simple: we only have one semantic model, and all modes of interaction are just different ways to modify it. Whether typing code to add a new child to some element, reparenting another via drag-and-drop, or describing the change in natural language via the AI assistant — all changes go through the same pipeline, and update the same model. And mode switching then becomes as simple as focusing on the desired window, optionally minimizing or closing the previous one.

[1] Internal pre-development survey, n=10, Jan 30 – Feb 2, 2026.

Hence the first problem is resolved: context is preserved, while all three approaches are enabled at the same time.

The second problem — scale management — required conceptually different approach. Now, for filtering, folding, coloring, session management, and similar concerns, we need to store it all in a single place, yet the model should not be modified every time we look at it differently. For this reason, a separate model was created — the **`ViewState`**. It allows user to hide or dim all elements that are currently unused, fold elements on the deeper levels, try out different layouts — all while preserving the original model.

The third problem — static nature of existing diagrams — was constructed because of the belief that the diagram that seems like a system provides significantly less value than one that behaves like one. The custom execution engine allows users to run diagrams both observing how elements move between nodes using the defined flows and decision points, and validating the design.

Section 5.5 revisits these same three problems through diagrams that other people actually built with DiagraVinci, on their own projects, independently of this thesis.

1.3 Objectives

Objectives of this project were the following:

1. Implement live bidirectional editing via textual and visual modes, ensuring any changes made in one are correctly reflected in another in real time.
2. Implement the AI code generation via text prompts in plain English with possibility to extend the existing diagrams as well as create new ones.
3. Build custom DSL with minimal range of distinct elements and relationships, each conveying a different semantic meaning, without limiting users in ways of combining them.
4. Implement a custom execution engine that allows users to run diagrams based on specific rules that require no separate syntax from what is used for creating ordinary diagrams.
5. Implement the system of boolean selectors that allow dimming, hiding, and coloring elements based on the user-defined rules, allowing to focus on specific parts of the system.
6. Ensure this is fully offline, client-based application with no dependency on a server.

1.4 Method

The project was developed in 4 sprints: discovery (research and requirements), foundation (parser, bidirectional sync), features (execution engine, filter system, templates, AI), and validation (testing, CI/CD, user testing).

The implementation follows 4-layered clean architecture, with Presentation, Application, Domain, and Infrastructure. Supplementary project — a design system library **`c7one`** — was developed in parallel. It provides the shell of the application: static and dynamic windows, mobile infrastructure, live styling and customization from layout to custom properties to theming and beyond, a repository of common components, centralized configuration, and more. Published as an **`npm`** package and used as a dependency.

1.5 Structure of this Report

Section 2 overviews the landscape of existing tools, explores the gaps and the requirements that stem from them

Section 3 provides an overview of the architecture, primary decision of separation of **`DiagramModel`** and **`ViewState`**, synchronisation pipeline, DSL design, tech stack, and the **`c7one`** library.

Section 4 goes in detail on the implementation of each subsystem: parser and code generator, bidirectional synchronisation manager, execution engine, selectors system, AI integration, tab synchronisation, persistance, and CI/CD.

Section 5 goes over the test suites, methodology, user testing and results, and summary of current bottlenecks and limitations.

Section 6 summarises results, compares to initial goals, reflects on the attained experience, and suggests possible directions of future work.

2 Research

In this chapter I perform an overview of the existing tools landscape to highlight the gaps present that serve as a motivation for building DiagraVinci. Particularly, we will explore 4 main categories: visual editors, code-based tools, AI-powered applications, and such that provide the functionality of execution. For each tool asking 3 main questions: which approach does it prefer working in? what are the benefits of these tools? and what are the limitations?

Contents

2.1 Traditional Visual Editors

WYSIWYG editors — drag boxes, draw lines — default choice for many

2.1.1 Lucidchart [1]

One of the most popular and most polished tools of the kind. Real-time collaboration, integration with Google Workspace and Atlassian, big template collection. If the team uses Confluence on daily basis, Lucidchart fits here naturally.

What it cannot do: convert diagram to code, version control along with codebase, filter, fold, switch between different views of the same model. Most of the genuinely useful functionality is only accessible on paid plans.

2.1.2 Draw.io [2]

Open-source project, free, offline, no vendor lock-in. All diagrams are saved as XML, which technically can be version-controlled. Large library of elements, a lot of control over visual aesthetics.

The XML format is not designed for manual editing, and the tool itself does not provide any features to make it easier. No code generation, no execution. Collaboration is possible, but as a workaround — users must share the file and coordinate edits manually.

2.1.3 Miro [3]

Infinite canvas that is great for workshops, brainstorms, design-sprints. Diagramming is just one of the many tools it offers. This can be seen as both pros and cons, as it allows wide range of tools, but is not optimized for documentation. Lacks functionality of diagram-to-code and execution.

2.1.4 Common for Visual Editors

Diagram, made within any of those, can convey a lot of information to the users — upon seeing “Database” rectangle or icon user can immediately understand the intent and relationships to the other components. Unlike the tool itself, which sees all elements as just shapes, and all relationships as just lines. All meaning live in the users’ interpretation, without possibility of validation or execution. All diagrams are visual images, so even small change might require manual repositioning of all nodes, and versioning and codebase storage requires workarounds when needs to be implemented.

2.2 Code-Based Diagram Tools

The paradigm is opposite: the diagrams are defined by code fully. Can work exceptionally well if version control is needed or if integrated into existing process.

2.2.1 PlantUML [4]

Native for git — text files can be well version controlled and reviewed via code review. Plugins exist for different IDEs. Same inputs always produce same outputs.

Cons: no visual editing. Complex diagram means huge unreadable text. Often also huge convoluted diagram with no possibility of improving the layout.

2.2.2 Mermaid [5]

Not only allows git versioning, but is also supported by many tools like GitHub, GitLab, Notion, Obsidian, without even plugins. Syntax is simpler than PlantUML. Browser-based, no server required, grows in popularity.

Cons: static SVG as an output, no visual interactivity. Limited syntax with shallow nesting.

2.2.3 Common for Code Editors

Valued for git compatibility, but provide no possibility of visual editing of the generated diagrams. Require knowledge of the syntax, often pose specific limitations. Become hardly manageable when scale grows.

2.3 AI-Enhanced Diagramming Tools

The newest category, the simplest approach, the least control.

2.3.1 Eraser.io [6]

Natural language to technical diagram in a few clicks: allows to receive a diagram by entering a prompt like "Create a microservice architecture with API layer, user service, and PostgreSQL database." Understands terminology, trained on extensive amounts of data. DSL similar to mermaid. Supports collaboration.

Cons: users are fully bound to the custom syntax. Updates to the diagram do not highlight differences, requiring manual identification of all changes. The final result is still a static image.

2.3.2 Whimsical AI [7]

Easy and fast generation of a diagram. Great for a specific range of diagramming — product flows, wireframes — less so outside.

Cons: no code presentation, limited diagram type selection, static diagrams.

2.3.3 ChatGPT with Diagram Plugins [8]

Convenient, familiar, and flexible. Allows transitioning to and from diagrams mid-conversation, capturing more context or helping with supplementary materials.

Cons: is not a diagramming tool per se, but rather a byproduct of code generation. Results might be inconsistent, manual editing inaccessible. Works very poorly with large diagrams.

2.3.4 Common for AI Tools

The most intuitive and very convenient approach. Often biased towards trained data, no semantic validation, limited manual interaction. And has a huge problem with the work on scale.

2.4 Execution and Simulation Systems

Tools that provide more than just static images

2.4.1 MATLAB Simulink [9]

Diagrams are executable models. Vast domain libraries for aerospace, automotive, and engineering systems. Code generation from models for embedded systems.

Cons: narrow specialization on mathematic modelling of physical systems. Expensive, requires full ecosystem MATLAB. Not the tool for diagramming application architecture.

2.4.2 AnyLogic [10]

Multi-method simulation software for logistics, manufacturing, healthcare. Strong visualization and optimization.

Cons: not designed for software architecture modelling. Steep learning curve, expensive licensing.

2.4.3 Common for Execution Tools

Very powerful for specific domains, but too domain-specific and heavyweight, with a high learning curve. Unusable for general-purpose diagramming or designing application architecture.

2.5 Gap Analysis

Three most important gaps — related, but distinct in essence.

2.5.1 Gap 1: Input Limitation

Each tool limits to just one input format. Visual to the mouse, textual to code, and AI to plain language and custom syntax.

The survey concluded that the most popular workflow contains multiple input modes, depending on the task. None of the tools supports it on the level of a single model. Every switch means a different tool, another format, rework needed.

2.5.2 Gap 2: Scale Collapse

All tools degrade after a few dozens of elements. Neither supports filtration based on some specific criteria, folding based on levels, or different views.

2.5.3 Gap 3: Diagrams That Never Move

All diagrams are like static images with no possibility to see how elements would flow and interact. No possibility to validate systems interactively or observe their behaviour. And for diagrams as tools whose primary goal is to help understand the system, execution cannot be underestimated.

2.6 Competitive Positioning

DiagraVinci does not compete on any of the strong sides of the tools that exist. Instead, it fills in the gaps that are present in diagramming today. DiagraVinci attempts to solve the problems that exist across diagramming tools by providing tri-modal input, selector system, and diagram execution.

Tool	Visual	Code	AI	Execution	Scale mgmt
Lucidchart	+	x	partial	x	x
draw.io	+	partial	partial	x	x
Miro	+	x	partial	x	x
PlantUML	x	+	partial	x	x
Mermaid	x	+	partial	x	x
Eraser.io	partial	+	+	x	x
Whimsical AI	+	x	+	x	x
DiagraVinci	+	+	+	+	+

Table 1: Capability comparison. “partial” means limited or one-directional support. Scale management covers filtering, folding, and multi-view layout.

2.7 Conclusion

Three major gaps: input modes are dispersed between different tools; all degrade when scale grows; diagrams are static images. DiagraVinci sets out to resolve all three issues.

3 Design

The three gaps identified in the previous chapter now get transformed into ten requirements, which, in turn, then build into the existing architecture that satisfies them. This chapter will review the requirements, the resulting architecture, and the decisions that were made.

Important to note that as the purpose of the tool is to help tackle the problem of complexity and project-long diagramming support, I made a decision to construct a framework that would help achieve this. Its purpose is to lay out a very primitive yet scalable approach to use DiagraVinci from start to finish.

Contents

3.1 DiagraVinci as a Framework

Whilst most tools treat diagramming as a single point in development that can be roughly described as “open - diagram - close”, DiagraVinci sets out to provide users with a framework that treats diagramming as a continuous process and suggests the respective handling. This framework consists of four stages — sketch, deepening, showcase, and evolution — that restart the loop by re-envisioning the first stage.

The purpose of the framework is to make the whole process as simple and straightforward as possible. It also helps make sure users get the most out of the tool, while doing as much as they feel comfortable with. And it is also to map out all features in a way that clearly states their purpose and place within the whole.

The four stages are as follows:

3.1.1 Sketch

The very first step in the process of building any system. Whatever architecture, it is much harder to be explained or improved until it can be seen. Even a rough sketch provides much more value than no diagram at all. On this stage, users have four main options that can be used alone or combined: code input, visual interaction, AI assistance, and template library. Each one has its own benefits: canvas for spatial thinking, code for auto-layout, templates as starting points or proven solutions, and AI for plain-language editing.

3.1.2 Deepen

The next step is adding details. Similarly to the previous step, users are not limited in how much detail they can include, and one of the reasons is that the process is iterative, and allows adding details incrementally. On this stage we complete the diagram to whatever level of detail we find useful at that point. We add elements, relationships, labels, flags, or anything else worth noting. Semantically it can mean adding properties, methods, interfaces, input, output types, dependencies, flows, named relationships, and so on. Complete syntax explanation is provided to user for reference.

3.1.3 Showcase

The step after this is when we finally get to prepare this diagram to be reviewed. This step is optional, but it is also very helpful when showcasing the diagram, as it helps to understand it better, to more easily focus at the parts where needed or to easily switch between different areas. The usage ranges from something as simple as coloring different element types differently for better contrast, to creating different sessions each with specific filters and colorings to allow to work on each area separately, while keeping SSoT. It can help present the diagram, and it can help work on it. The system of selectors and sessions exists exactly to help with this: color, dim, or hide elements by type, by name, by depth, or flags; highlight the group that takes part in a specific flow, while dimming the rest; save this view as a named session and switch between ‘developer’ and ‘manager’ modes in a single click; and so on.

This step also includes the usage of the diagram execution, as it is also of a very high value during a presentation. It allows showing how elements move following specific flows, decision points, filtering, routing, or some other specific traversal logic. This presents the real behaviour of the diagram in an evident way.

3.1.4 Evolve

And the final step is the evolution of the diagram. The tools are the same as in the first step, but now repurposed to be used on a fully functional diagram as the next iteration of its development. Now templates window serves as a repository with snapshots, letting users save the current version of the diagram in the corresponding collection; the visual, textual, and assisted editors are all available to sketch out the next features or improvements to the existing ones. Flags can be added to mark future changes or the scope of changes in an iteration. Diffing allows comparing different versions (i.e., saved snapshots) of the same project. Architectural templates can be explored, or AI suggestions requested on how the diagram can be improved or fixed.

The framework is nothing more than a mere suggestion for simple and efficient use of the tool.

3.2 Requirements

The gaps from before are mapped to these 10 functional requirements:

ID	Requirement	Gap
R1	Visual LCNC editor: element and relationship canvas manipulation, properties panel, shortcuts	G1
R2	Code editor: element and relationship definition, shortcuts	G1
R3	AI generation: natural language to diagram, context usage possibility	G1
R4	Live bidirectional synchronisation between textual, visual, and AI modes	G1
R5	Filtering: selector-based system allowing hiding or highlighting elements based on specific conditions	G2
R6	Sessions: saved groups of selectors set to specific modes for easy mindset switch	G2
R7	Layouts: a range of different algorithms to select the preferred layout	G2
R8	Templates: built-in collections of templates, possibility to add, save, and export custom ones	G2
R9	Execution: token flow through predefined paths, with specific behaviour based on elements	G3
R10	CI/CD pipeline: automatic linting, testing, predeploy, and deploy on each pull request	—

Table 2: Core requirements based on the gap analysis. Gap column: G1 = input fragmentation, G2 = scale collapse, G3 = static diagrams.

Non-functional requirements include: fully offline, client-based, deployed as static website, completely free, no account necessary, everything created by user can be exported and imported without limitations.

3.3 Architecture Overview

DiagraVinci follows a four-layer clean architecture [11]:

- **Presentation Layer**: react components, responsible for presentation and user interaction. Contains no business logic — only observing events and sending redux actions.
- **Application Layer**: orchestration and state. **`SyncManager`** coordinates all model updates. **`ExecutionEngine`** manages simulation. Redux store serves as a SSoT.
- **Domain Layer**: core business logic with no framework dependencies. Models, layout algorithms, selector evaluators, synchronisation utils. This level can be tested without a browser.
- **Infrastructure Layer**: all infrastructure that runs the process and input/output with the external systems. Includes lexer, parser, code generator, gemini client, indexDB storage, import, export, and more.

All changes, regardless of the origin, go through the **`SyncManager`** before being saved to storage.

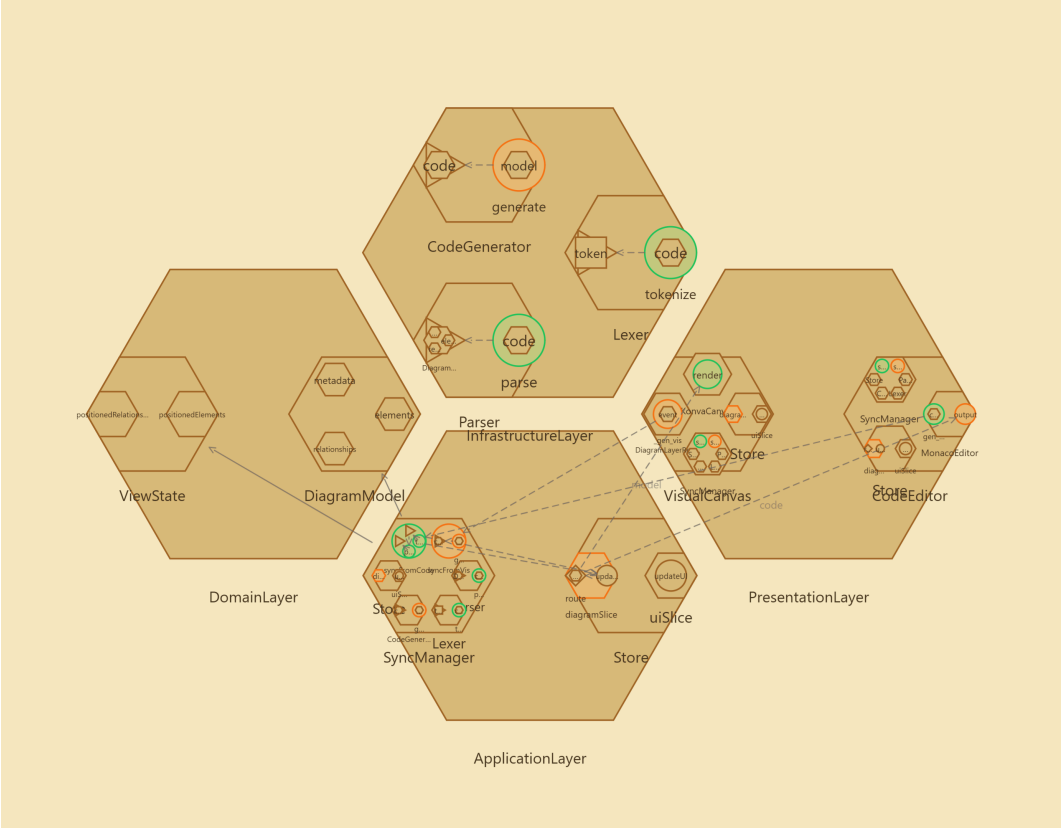


Figure 1: DiagraVinci core architecture — the synchronisation pipeline and its three input modes. Full codebase structure is described in the Implementation chapter.

Alongside this diagram, file-to-flow relationships are tracked in a notation devised for this project — a "codiagram": a per-file table of which user-facing flow (code edit, visual edit, UI interaction, AI generation, execution, multi-tab sync) touches which source file,

and roughly in what call order. A whole number marks a step that hands control to a different actor — a Redux dispatch, a callback into another subsystem; a floating-point number marks a synchronous call that returns to its caller before the flow continues. The legend, a worked example, and the full file table are in Section AC.

3.3.1 Technology Stack

Technology	Role	Why
React 19 + TypeScript 5.9 [12]	Presentation layer	mature and popular ecosystem, well-fitting unidirectional flow for bidirectional sync, static typing, catching errors early
Konva.js + react-konva [13]	Canvas rendering	HTML5 Canvas for efficiency on huge diagrams; Konva has tooling for event-handling and grouped actions
Monaco Editor [14]	Code editor	keyboard shortcuts, hints, highlights, convenient editing experience with minimal setup
Redux Toolkit [15]	State management	Redux DevTools are great for troubleshooting state synchronization
Google Gemini 2.5 Flash Lite [16]	AI generation	Daily free requests, easy setup in a few clicks, accessible to everyone
Vite 7 [17]	Build tool	native HMR for ES modules; fast feedback loop during development
Tailwind CSS 4	Styling	Removes the need for separate css files per each component
@levkobe/c7one	UI shell	application shell, with dynamic panels, customizable layout, theming, mobile layout, i18n, and component library
Vitest	Testing	vite-native test runner; fast for this stack
GitHub Actions [18]	CI/CD	Free, github-native, allows linting, testing, predeployment, and deployment

Table 3: Technology stack and explanation.

Rejected alternatives: Vue 3 (smaller ecosystem), Svelte (no prior experience, introducing high learning curve risks), WebGL/PixiJS (over-engineered for static diagramming), ANTLR/PEG.js (learning curve exceeds benefit at this DSL complexity), Zustand (unavailability of Redux DevTools), Vercel (GitHub Pages is simpler and has no drawbacks).

3.4 Domain Model: Two Representations

The most impactful decision within this project scope was to store semantic elements and their visual representation as separate data structures.

`DiagramModel` — semantic truth — contains elements (with type, name, child elements), relationships, selectors.

`ViewState` — visual state — contains positions, folded, hidden, and dimmed elements, zoom, pan.

This split is justified by:

- **Layout independence** — the model does not change based on its visual representation, hence switching from circular to hierarchical layout should only modify visual data.
- **Camera state** — the model does not need to store any information about the current zoom and pan, so when saved no unnecessary data is stored
- **Multiple positions** — an element might be mentioned within the same diagram multiple times, and to not redefine the children each time, element's definition is single, but positioning can be multiple
- **Saving diagram** — when saving only the code of the diagram, all visual data is redundant, so only **`DiagramModel`** needs serializing

3.5 Synchronisation Architecture

DiagraVinci has 3 change sources: code editor, visual canvas, and AI panel. All three work via the same flow.

Step	What happens
Input	A new word entered in code editor or by AI, or interaction with the canvas
Parse or generate	**`Code`** -> **`Lexer`** -> **`Parser`** -> **`DiagramModel`** for code, **`DiagramModel`** -> **`CodeGenerator`** -> Code for visuals
Diff	**`ModelDiffer.diff(old, new)`** -> set of added/removed/updated elements
Early exit	If **`ModelDiffer.isEmpty(diff)`** -> only **`setCode`**, skip recompilation
Merge	**`ViewStateMerger.merge(diff, oldVS, newVS)`**
Dispatch	**`dispatch(syncDiagram(...))`** -> storage is updated, React rerenders

Table 4: Synchronization flow: common structure for different input modes

3.6 DSL Design

The DiagraVinci DSL design was ideated before implementation. The idea was to achieve the syntax that is very simple, intuitive, flexible, and scalable, and on top of that also grants the semantic meaning to all elements and relationships — enough to represent any common diagram type, but not more than what would feel overwhelming.

The six element types that exist in the DSL are:

Syntax	Type	Semantic Role
`name{}`	Object	Classes, entities, modules, components
`name[]`	Collection	Arrays, groups, collections, queues
`name||`	State	State, status, lifecycle stage
`name()`	Function	Operations, methods, processes, steps
`name>>`	Flow	Data streams, I/O, pipeline segments
`name<>`	Choice	Conditional branches, decision points

Table 5: DiagraVinci element types. The wrapper character encodes semantic role.

The six relationship types are:

Syntax	Name	Semantic Role
`a --> b`	Association	A standard directed dependency
`a ..> b`	Flow	A data or execution flow between elements, rendered dashed
`a --|> b`	Inheritance	IS-A or extends
`a ..|> b`	Realization	Implements a contract
`a o-- b`	Aggregation	Has-a with weak ownership
`a *-- b`	Composition	Part-of with strong ownership

Table 6: DiagraVinci relationship types. Labels can be added by wrapping words between arrows like so: `--label-->`.

What makes it a distinct language rather than a formatter configuration:

- **Names are references** — each name refers to the same exact element, so reusing it will carry all element's contents
- **Minimal syntax** — default element type does not require wrappers, relationships do not require prior element definition
- **Stable cyclic transformation** — code entered by user, and generated from canvas, give the same `DiagramModel`. The code generator always produces code deterministically.

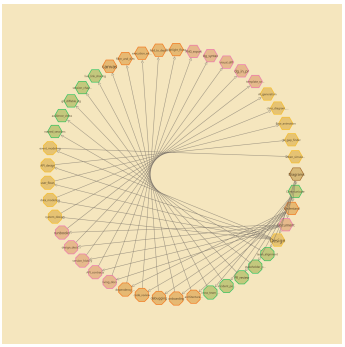
3.7 Layout System

Layouts in DiagraVinci are interchangeable. All of them extend `BaseLayout`:

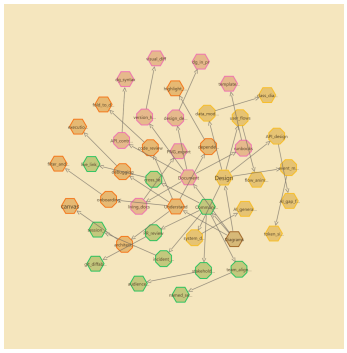
Algorithm	Best For
Basic / Circular	Default starting point; radial arrangement weighted by connectivity
Hierarchical	Tree structures, hierarchies; topological sort into vertical levels
Radial	A central concept with outwarding directions; concentric rings by distance
Timeline	Sequential flows, execution order; left-to-right by depth level
Pipeline	Source/process/sink data flows; parallel horizontal lanes
Force-Directed	Organic element placement; spring physics
Execute	Execution simulation; driven by **`ExecutionEngine`** tick output
Manual	User-defined positions with no automatic repositioning

Table 7: The eight DiagraVinci layout algorithms.

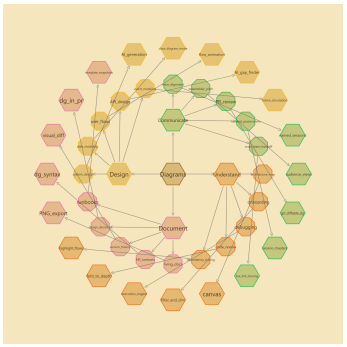
Nested elements can be accessed via dot syntax, e.g., **`Auth.login`** is the path to **`login`** nested inside **`Auth`**. Each element can have different positions, and relationships choose the shallowest element if no concrete path is provided.



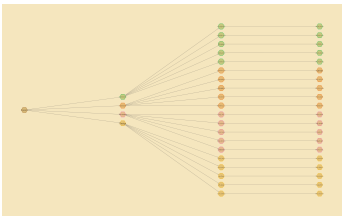
Circular



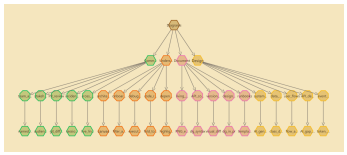
Force-directed



Radial



Pipeline



Hierarchical

Figure 2: Five of the eight layout algorithms applied to the same diagram.

3.8 c7one — The Supporting Design System

The shell for the application is provided by my own npm library **`@levkobe/c7one`** that I have developed in parallel and published on GitHub Packages.

The primary motivation leading to it was that all my projects share the same ideology, often the same approaches, they solve the same problems, and they all would benefit from reusing common behaviour. The library is built around different C's:

- **Configuration**: All design expressed as data — tokens, themes, params
- **Centralization**: One provider controls all visual state
- **Customization**: Every value is adjustable — by developers and by users
- **Composition**: UI built from reusable, layered primitives
- **Consistency**: Shared tokens ensure visual coherence across all apps
- **Control**: Full runtime read/write access via a unified API
- **Canvas**: A layout system for structural composition (different windows)

What DiagraVinci uses from c7one:

- **`AppShell`** — high-level shell with fixed header, full-screen canvas, and floating layer of adjustable windows that transform into tabs on mobile
- **`DynamicPanelRoot`** — a system of panel management. Each one (code editor, canvas, AI panel, properties, selectors, templates, etc) is registered as a window that users can open, close, minimize, or adjust based on their preferences.
- **`C7OneProvider`** — introduces all design tokens such as css variables on the **`:root`**
- **`SettingsPanel`** — interactive interface for setting color theme, css settings, or developer-defined parameters
- **`AppConfigProvider`** — application configuration (all color, size, scale, and similar configurables)
- Component library — buttons, input fields, modals, badges, etc.

3.9 Conclusion

The decisions that have the most impact on the architecture:

- **`DiagramModel`**/**`ViewState`** split — enables reusable elements, creates clear separation of concerns, and speeds up visual changes,
- Single pipeline running through **`SyncManager`** — ensures consistent handling regardless of input mode
- Custom DSL — minimal syntax with semantic meaning

4 Implementation

The project was implemented in four sprints: requirement analysis, foundation (parser and synchronization), functionality (execution engine, layouts, filters, templates, AI), and validation (tests, CI/CD, user testing). This chapter provides an overview of the most important and interesting decisions and their implementation.

Contents

4.1 Parser and Code Generator

Parsing code is arguably one of the most important processes in the chain. If there is a mistake within it, all the subsequent steps will build on incorrect results, producing either wrong diagrams, or nothing at all. Parsing contains these elements:

Lexer (**`src/infrastructure/parser/Lexer.ts`**): reads entered text, and splits it into a list of typed tokens, such as **`NEWLINE`**, **`RELATIONSHIPS`**, **`OPENING_WRAPPERS`**, **`CLOSING_WRAPPERS`**, or words-identifiers.

Parser (**`src/infrastructure/parser/Parser.ts`**): recursive-descent parser over the list of input tokens. The recursive entry point is **`parseContents()`**, with **`lastEl`** and **`lastRel`** tracking the last elements and relationships to enable chains like **`a --> b --> c`**, or element type definition via subsequent wrappers following the element. The parser builds the **`DiagramModel`** directly.

Code Generator (**`src/infrastructure/codegen/CodeGenerator.ts`**): an inversion of the Parser: **`generateElement()`** is the recursive entry point. **`AncestryTracker`** tracks nesting and identifies loops. The resulting code is deterministic, with currently only one formatting preference.

All the errors are silently swallowed, although the syntax is very flexible, so such errors are very rare.

4.2 Bidirectional Sync

`SyncManager` (**`src/application/SyncManager.ts`**) is the central class of the Application layer. It has three entry points:

```
syncFromCode(code: string): void  // user typed in editor
syncFromVis(model: DiagramModel): void  // user interacted with canvas
syncFromAI(code: string): void  // AI generated a diagram
```

The optimizations, one in the form of an early exit on an empty diff, and another that fully avoid **`SyncManager`** on simple repositioning of an element, both remove significant part of model recreation hence improving the performance.

4.3 Execution Simulation

The execution engine (**`src/application/ExecutionEngine.ts`**) is one of the features that significantly differentiate DiagraVinci from traditional documentation tools.

Main abstraction here is the **`ExecutionStepData`**: the description of what has changed in one tick of a simulation.

Every tick dispatches different events based on the element's behaviour:

- **Object**: Upserts the token's children into its own — same-named children replace existing ones, new ones are appended.
- **Collection**: Accumulates the token; it persists at this element.
- **State**: Accumulates the token; it persists at this element.
- **Function**: Receives the token as input, using its children as arguments.
- **Flow**: Replaces the token's payload with clones of its children; passes through unchanged if empty.
- **Choice**: Routes to the "yes" branch (or the first branch if none is labeled "yes") when a child condition matches; to "no" otherwise. Routes all to "yes" if no children.

The tool supports a few special functions that can perform specific actions that correspond to their names: **`gen()`** (generates new elements each tick), **`round_robin()`** (routes), **`multiplier_N()`** (duplicates specific quantity), **`duplicator()`** (duplicates to all outgoing relationships), **`throttler_N()`** (swallows), **`connector()`** (creates relationships), and **`disconnector()`** (breaks relationships).

The execution engine operates on the **`DiagramModel`**, essentially creating new diagrams upon execution, and these diagrams can be materialized or reset to original.

4.4 Filter and Selector System

Filtering, folding, dimming, and coloring all generalize into one system: groups. A group matches elements by a **`regex=`** pattern against the element's full type-embedded path (e.g. **`game{}.player{}`**), by a **`compose=`** boolean expression over other groups, by an attached **`:flag`**, or any combination. It contains the following classes:

- **`GroupExprParser`** (**`src/domain/selector/GroupExprParser.ts`**) — parses the **`compose=`** expression: **`&`** (and), **`|`** (or), **`-`** (and-not), with parentheses for precedence.
- **`GroupEvaluator`** (**`src/domain/selector/GroupEvaluator.ts`**) — decides whether a given element matches a given group, combining its regex, compose, and flag checks.
- **`SelectorEvaluator`** (**`src/domain/selector/SelectorEvaluator.ts`**) — a narrower evaluator for the one case **`GroupEvaluator`** doesn't cover: turning a click-selection on the canvas into a group, by generating its regex directly from the selected elements' paths.
- **`SelectorPresets`** (**`src/domain/selector/SelectorPresets.ts`**) — built-in starting groups available before the user defines any of their own.
- **`FilterResolver`** (**`src/domain/sync/FilterResolver.ts`**) — composes the per-mode path lists (**`hiddenPaths`**, **`dimmedPaths`**, **`coloredPaths`**) from whichever groups are active in the current session. **`dim`** and **`hide`** are inverted: a group names what to keep, and **`FilterResolver`** works out everything that's left over.

Sessions assign a mode — **`color`**, **`dim`**, **`hide`**, or **`off`** — to each group at once, for one-click perspective switching. Earlier **`!rule`** and **`!selector`** directives are still accepted and silently migrated to **`!group`** on first save.

The **`filterSlice`** includes a **`_rev`** counter that increments on every change. **`VisualCanvas`** subscribes to this counter and updates whenever it changes.

4.5 AI Integration

Pipeline: **`AIOrchestrator`** -> **`PromptBuilder`** -> **`GeminiService`** -> **`ResponseValidator`** -> **`SyncManager.syncFromCode`**.

`PromptBuilder` constructs the request. Three modes:

- **Generate:** system prompt with full DSL documentation plus user request. The output is new diagram code.
- **Extend:** current diagram code, then instructions plus user request. The output is the updated diagram code.
- **Analyze:** current diagram code, request to fix issues or suggest improvements. The output is textual response from the AI.

`ResponseValidator` first runs the parser on the new model, and only accepts the response if no errors appear. The limitations are based on the user's Google Gemini plan. API key is stored locally, in **`localStorage`**.

4.6 Multi-Tab Synchronisation

`TabSyncManager` processes synchronization between tabs via Web API **`BroadcastChannel`** [19].

Handshake **`HELLO`**/**`HELLO_REPLY`** solves the issue of **`BroadcastChannel`** having no message history. So the new tab sends **`HELLO`** and the existing send **`HELLO_REPLY`**, completing the handshake successfully.

All the structural changes are synchronized. But data like active selectors is not, which allows having different tabs opened with different views of the same diagram.

4.7 Persistence and Infrastructure

4.7.1 Storage

State lives in three places, by how long it needs to survive. **`DiagramModel`** itself is canonical in the code — **`.dg`** text, written by **`CodeGenerator`** and restored by **`Parser`**. Everything visual (**`ViewState`**, group/session state, UI preferences) is written by **`persistence.ts`** into IndexedDB, with **`localStorage`** as a synchronous fallback, debounced 1 second after every change — this is what survives a page reload. A third layer is intentionally short-lived: which session is active lives in **`sessionStorage`**, so different tabs can sit on different sessions, while selection state and undo/redo history are in-memory only and reset on reload.

Saving a diagram is possible in two formats:

- **`.dg`** (syntax only): Minimal, convenient, useful for version control
- **`.jsonl`** (full snapshot): contains both **`DiagramModel`** and **`ViewState`**. Recreates the diagram fully, preserving all positional data as well as camera state.

4.7.2 Template System

Templates are the diagram code snippets that are grouped into **`TemplateCollection`**s.

`CollectionRepository` downloads all built-in and user-defined collections upon start. For export/import collections are zipped using **`fflate`**.

Templates can be added, removed, exported, and imported, all via interface on Collections window.

4.7.3 Deployment and Live Demo

The application lives at **`https://levkobe.github.io/diagravinci/`**, where it is deployed by GitHub Pages. All code is available at **`https://github.com/LevkoBe/diagravinci`**.

4.7.4 CI/CD Pipeline

On every pull request and push to **`main`**, GitHub Actions run the following:
- Strict TypeScript typechecking
- Linting with ESLint
- Unit and integration tests with Vitest
- Vite production build

When PR is created, coverage report is generated, and a deployment is made at a distinct address.

When PR is merged into **`main`**, all the changes are automatically deployed to the original link.

4.8 Conclusion

These were the main decisions and their implementation. Some were taken immediately, while others evolved gradually into the system I have now. They might still evolve later, but the core will likely remain.

5 Validation

This chapter goes over all three types of testing that was used: automated test suits, a structured user test guide that was applied to live deployed version of the application, and CI/CD test gates. It also provides their assessment and lists existing limitations and issues of the application.

Contents

5.1 Test Strategy

DiagraVinci uses **`vitest`** as the test runner and React Testing Library for component testing. Overall, the testing includes:

- Unit tests: check all modules separately. Parser might be tested the most, as it was one of the first to be tested, and also because since it was developed, many bugs or uncertainties were discovered, each one adding new test cases.
- Integration tests: verify common work of a few modules together. Most exemplary instance here is likely layout + rendering tests. But also, for reasons of convenience, these include tests on token + parsing + action, as it is often more understandable and applicable.
- Manual tests: document scenarios for testing that cannot be covered automatically in the same way, like AI generation, drag-and-drop functionality, or the like.

CI/CD pipeline starts tests on every pull request update.

5.2 Parser Tests

Test suite includes:

- All element types, with and without children, named or anonymous
- All relationship types, with and without labels, chained and not
- Nesting of different depth, reference to the same element from different places
- Edge cases, like empty entry, space characters, or comments
- Syntactically incorrect inputs with partial models as an output

5.3 Layout and Render Tests

Tests on layout algorithms verify element positioning — each element should have one, they should not be the same, they should fit the boundaries, should be correctly positioned relative to each other.

Render tests validate that expected Konva shapes appear on the page having specific state. Navigation tests check that arrow navigation works properly. Execution tests check correct state at particular execution tick.

5.4 User Testing

A structured guide (**`USER_TESTING.md`**) was created and shared, containing a list of suggested exercises to complete with or without a general video tutorial.[2] A Google Form was also provided for feedback on the application.[3]

The guide contained:

1. Visual diagram creation — place a few elements by click, connect them
2. Text editor changes — create a simple diagram via code
3. Layouts — apply different layouts to the same diagram and review them
4. Templates — explore and apply some templates, edit the diagrams
5. Execution simulation — execute the template from the previous step or from provided code
6. Filtering and highlight — create a selector and apply dimming mode
7. AI generation — generate a diagram using AI panel, update it

5.4.1 Results

The survey received 6 responses. Of these, 2 were software engineers, 2 were students, one was a designer, and one a teacher/researcher. Experience levels varied from rare use to frequent, with most using diagram tools occasionally.

Overall experience at the moment of survey was rated from 2 to 5 with average of 3.5, with those having prior experience in diagramming rating their experience higher, while those with less experience were initially disoriented.

Positive feedback included:

- Smoothness of the interface and Performance
- Nesting mechanic — found useful and convenient
- Integration with Gemini AI, templates, dark theme
- Nice animations for execution

Issues found by users:

- Light theme was visually monotonous — panels, controls, and canvas — all blended together
- Built-in templates were not obvious for new users, making first interactions with tool even harder
- The purpose of the ‘Properties’ panel was unclear to some users
- Toolbar contents overflowed component boundaries upon hovering
- Mobile version was not working
- Execution was not running the simulation properly
- Elements are repositioned automatically when not needed
- Diagram is easy to lose, viewport navigation not placed conveniently
- Some buttons lack clear names or tooltips

[2] Full walkthrough: youtu.be/qvlEvPjr9JI. Following the Sketch–Deepen–Showcase–Evolve framework end to end, from a blank canvas: youtu.be/IpJ8uAoZsFw.

[3] User testing feedback form (always active): forms.google.com

The feedback was taken into account and many changes were implemented successfully, including:

- More contrasting and clearly separated windows
- Toolbar redesign for simplicity and convenience
- Mobile version improvements
- Execution simulation fixes
- Layout application limited to cases when repositioning is necessary
- Tooltips, labels, and icons added where appropriate

Many other changes are in the backlog to be added when time permits.

5.5 Real-World Use Cases

Outside of the structured testing guide, some of my classmates tried DiagraVinci for their own projects. This time the focus was on using the application for something real, rather than working through a tutorial. The gaps from Section 2 show up across these cases, to varying degrees.

5.5.1 Recycling-App Architecture

One classmate generated the cloud architecture for a recycling-sorting app directly from a structured prompt in the AI panel — an iOS client, AWS Cognito authentication, an API Gateway, three backend services, and DynamoDB — then refined the result by hand, in both code and canvas. This is what Gap 1 is about: starting from a prompt and finishing by dragging boxes, on the same model, without exporting anything in between. When asked which features mattered most for this case, the author named tri-modal editing, presentation mode, AI generation with templates, switching layouts, and diff/merge.[4]

5.5.2 Feed-Ranking Pipeline

A second diagram modelled a dating-app feed-ranking flow as a pipeline — deduplication, mutual-match filtering, preference filtering, proximity filtering, into a ranked feed — laid out with the timeline layout, with each stage colored separately. A pipeline like this is easier to walk an audience through stage by stage than to explain from one static picture, which is what presentation mode and per-stage coloring are for.

5.5.3 Fitness-Dating Data Model

A third diagram modelled the data layer of a fitness-and-dating app: around eighteen entities — users, matches, chats, clubs, trainers, subscriptions, moderation reports — on one canvas. This is exactly the Gap 2 responsibility: a diagram this size stops being readable as a single flat picture, and is the kind of case the groups with coloring and filtering exists for, allowing to focus attention at specific elements, or visually differentiate between them without getting lost in scale.

5.5.4 Diagramming DiagraVinci Itself

The tool was also used to draw diagrams of itself: the architecture figure in Section 3, and every diagram used in the defense presentation slides, were authored in DiagraVinci. The

[4] Feedback gathered informally via direct messages with each diagram's author; available on request.

built-in template collection (Section 4) carries the same kind of architecture and design-pattern examples, for the same reason — using the tool to explain the kind of systems it was built to diagram. Three more self-diagrams — a use-case snowflake, a feature snowflake, and the framework cycle from Section 3 — are in Section AB.

5.6 Requirements Traceability

ID	Requirement	Validated by
R1	Visual LCNC editor: element and relationship canvas manipulation, properties panel, shortcuts	Rendering and navigation tests, task 1
R2	Code editor: element and relationship definition, shortcuts	Lexer, parser, code generator tests, task 2
R3	AI generation: natural language to diagram, context usage possibility	Manual testing, task 7
R4	Live bidirectional synchronisation between textual, visual, and AI modes	Unit and integration tests of parsing, visual interaction, and code generation
R5	Filtering: selector-based system allowing hiding or highlighting elements based on specific conditions	Selector unit tests, task 6
R6	Sessions: saved groups of selectors set to specific modes for easy mindset switch	Manual testing
R7	Layouts: a range of different algorithms to select the preferred layout	Layout unit tests, task 3
R8	Templates: built-in collections of templates, possibility to add, save, and export custom ones	Manual testing, task 4
R9	Execution: token flow through predefined paths, with specific behaviour based on elements	Execution unit tests, task 5
R10	CI/CD pipeline: automatic linting, testing, predeploy, and deploy on each pull request	GitHub Actions on each PR

Table 8: Requirements traceability matrix. All core requirements are implemented and validated.

All core requirements were implemented and validated, including execution simulation that was originally marked as optional.

5.7 Known Limitations and Open Questions

5.7.1 Relationship Ownership Ambiguity

Relationships behave differently based on the elements' existence. For instance, **`player o-- inventory`** if nested might connect the local player, or the one placed elsewhere.

The reason for this limitation is the clash of two requirements:

- chained relationships and implicit element definitions that allow conveniently declare relationship chains without the necessity of previous element declaration (e.g., compare **`a{b c a.b -- a.c}`** and **`a{b -- c}`**)
- and the possibility to define relationship to non-local elements (e.g., **`a b{c b.c -- a}`**)

To resolve the issue it is recommended to use dot notation for clear specification of the element being connected.

5.7.2 AI Semantic Accuracy

The AI generation is validated against errors, but this does not ensure the AI is creating the diagram with the expected intent, or that it preserves all elements that need to remain unchanged. Users can still easily revert changes, but currently diffing is not set up, and no control is provided over the changes the AI might implement. The diff/merge mode already built for manual code updates (Section 4) is not yet wired into the AI panel — Section 6.4 picks this up as the most direct fix.

5.7.3 No Real-Time Collaboration

`BroadcastChannel` synchronizes multiple browser tabs on one machine. But no server-based collaboration was implemented yet, as the tool is designed to be client-based. It is expected to change in the future, but as it was not the primary goal for this phase, all user changes remain local.

5.7.4 Syntax Semantics

The syntax for diagram definition, as well as execution semantics, was implemented following standard principles, common diagram types, and common sense. But as new features are added, syntax grows, and in some places it might benefit from redesigning to a more intuitive approach. Very likely candidates might be reusable templates, element metadata, modifiers, or else — these features are partially present in a way, but might seem counterintuitive to some. While any significant redesign is unlikely, the question of syntax remains open.

5.8 Conclusion

The test suites are set up to validate parsing, layout, rendering, execution, and more. Test guide provides overview of the core functionalities and steps to achieve concrete results. All the obligatory requirements are satisfied. The tool provides a range of secondary features, and has its known limitations and questions open to discussion.

6 Conclusion

6.1 Project Summary

DiagraVinci is a browser-based tool for diagramming, built around two concepts: complexity management, and continuous diagramming as an augmentation of the design process. Instead of avoiding complex diagrams, we should be able to work with scale in a flexible manner. And instead of creating diagrams as just snapshots of the current architecture state, diagrams are used throughout the development process, providing use on all stages of the project. This is achieved through the following capabilities of the tool:

- three input modes (visual, textual, and AI) with a single semantic model and live bidirectional synchronization
- selector system (dimming, hiding, coloring, folding) with boolean per-element evaluation that allows to focus on specific elements and save view presets as sessions
- interchangeable layout algorithms (hierarchical, circular, pipeline) that allow instant automatic layout based on the diagram type
- and execution engine (elements generation and movement) that allows to observe and validate the behaviour of the diagram

6.2 Comparison with the Initial Objectives

The initial goal stated: tri-modal editing on single model, live bidirectional synchronization, different semantic elements with distinct behaviour (e.g., in execution mode), filtering and folding functionality, fully offline free client-side application. The goal is fully achieved.

One substantial difference from the original plan is the deployment strategy. Initially, Vercel was expected to be used. But since all features were actually achievable within the same GitHub Pages, including predeployments, I defaulted to them. All functionality is preserved, with less effort.

DSL might differ slightly from the initial specification. For the most part, it was preserved, just extended with flags, dot notation, declaration blocks, semantic refinements, and other minor adjustments. The main syntax was fully implemented.

6.3 Encountered Difficulties

Likely the most troublesome was the implementation and multiple refactors of the bidirectional synchronization. While conceptually simple — three flows, one pipeline — in reality there are countless possible interactions the user might have with the diagram. Be it textual, visual, UI, or AI — it still has a range of possible actions within, each posing a risk of being an exception, like visual change causing another visual change, or similar in code. A set of non-obvious problems had to be resolved, like ensuring the repositioning does not trigger code regeneration, or if repositioned into another element, it should.

Another complexity appeared on the level of **`c7one`** library: any issues adjacent to library functionality, required attention and troubleshooting within both projects simul-

taneously. In retrospective, it might be beneficial to first design all UI requirements of DiagraVinci, and only then proceed to the library implementation.

Execution simulation was yet another complex process that required creation of a new model `ExecutionStepDelta` for description of one tick changes. Synchronization of the visual animations with the execution engine also required significant time investment. And so did the definition and implementation of distinct behaviours for all element types.

6.4 Future Aspirations

While the list of possible and hopeful additions to the project is already beyond finite, below are some of the likely candidates for nearest implementation:

- **Diagram lenses**: structural transformations that change diagram into dedicated type, like class relationships, flow, etc. Unlike the current system, this would strip diagram down to only the necessary elements, while adding semantic relationships were needed, and applying appropriate repositioning.
- **Architectural transformations**: architecture-based transformations that allow applying specific patterns and principles from the list of templates to specific parts of the diagram. For instance, inverting dependency by introducing interfaces. This would require user to select elements, transformation type, and click “apply” button.
- **Named flows**: currently navigation is based on the order of the relationships between elements. This feature would allow creating specific flows, that, like presentation slides, define a specific traversal order that could be used for a presentation.
- **Relationship selectors**: extension of the existing selector system by allowing to select based on the relationships, filter, color, or dim them, as well as elements that they connect.
- **Advanced layouts**: existing layouts feel off. They are stacked, often positioned in a way that relationships overlap or are positioned suboptimally. One of the ideas here is to use connectivity components to split layout into separate groups, freeing some space and positioning elements better relative to each other.
- **Advanced AI**: currently AI only is familiarized with the syntax, and has access to current diagram. The extensions might include interactive test generation, template browsing, general application knowledge and assistance, and more.
- **Input format extension**: extending input formats to different diagram syntaxes, or going even beyond, like, for instance, digestion of csv files.
- **Collaborative editing**: a feature of server setup, similar to how AI is absent, but all the infrastructure exists, making it trivial for users to add their own assistants. With functionality of creating separate rooms connected to the user-hosted server, that enables live collaboration without additional costs and with arbitrary scaling possibility.
- **Plugin system**: maybe the most powerful feature of all, it would enable custom plugins that could implement any and all of the beforementioned points, or experiment with something else, like new element or relationship types, layouts, visualizations, and more.

Glossary

Bibliography

[1] Lucid Software, "Lucidchart: Intelligent diagramming for every team." Accessed: Jan. 15, 2026. [Online]. Available: https://www.lucidchart.com/

[2] JGraph Ltd, "draw.io: Security-first diagramming for teams." Accessed: Jan. 15, 2026. [Online]. Available: https://www.drawio.com/

[3] Miro, Inc., "Miro: The online collaborative whiteboard platform." Accessed: Jan. 15, 2026. [Online]. Available: https://miro.com/

[4] A. Roques, "PlantUML: Open-source tool for creating diagrams from plain text language." Accessed: Jan. 15, 2026. [Online]. Available: https://plantuml.com/

[5] Mermaid Contributors, "Mermaid: Diagramming and charting tool for documentation." Accessed: Jan. 15, 2026. [Online]. Available: https://mermaid.js.org/

[6] Eraser, Inc., "Eraser.io: The whiteboard for engineering teams." Accessed: Jan. 15, 2026. [Online]. Available: https://www.eraser.io/

[7] Whimsical, Inc., "Whimsical: Design and collaborate faster." Accessed: Jan. 15, 2026. [Online]. Available: https://whimsical.com/

[8] OpenAI, "ChatGPT: Optimizing language models for dialogue." Accessed: Jan. 15, 2026. [Online]. Available: https://openai.com/chatgpt

[9] The MathWorks, Inc., "Simulink: Simulation and model-based design." Accessed: Jan. 15, 2026. [Online]. Available: https://www.mathworks.com/products/simulink.html

[10] The AnyLogic Company, "AnyLogic: Multi-method simulation software." Accessed: Jan. 15, 2026. [Online]. Available: https://www.anylogic.com/

[11] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017.

[12] Meta Platforms, Inc., "React: A JavaScript library for building user interfaces." Accessed: Mar. 01, 2026. [Online]. Available: https://react.dev/

[13] A. Parashchuk and Konva Contributors, "Konva.js: 2D canvas library for desktop and mobile applications." Accessed: Feb. 01, 2026. [Online]. Available: https://konvajs.org/

[14] Microsoft Corporation, "Monaco Editor: The code editor that powers VS Code." Accessed: Feb. 01, 2026. [Online]. Available: https://microsoft.github.io/monaco-editor

[15] Redux Contributors, "Redux Toolkit: The official, opinionated toolset for efficient Redux development." Accessed: Feb. 01, 2026. [Online]. Available: https://redux-toolkit.js.org/

[16] Google LLC, "Google Gemini API: Build with generative AI." Accessed: Feb. 10, 2026. [Online]. Available: https://ai.google.dev/

[17] Vite Contributors, “Vite: Next generation frontend tooling.” Accessed: Feb. 01, 2026. [Online]. Available: https://vitejs.dev/

[18] GitHub, Inc., “GitHub Actions: Automate your workflow from idea to production.” Accessed: Mar. 15, 2026. [Online]. Available: https://github.com/features/actions

[19] MDN Web Docs, “BroadcastChannel API.” Accessed: Apr. 01, 2026. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/BroadcastChannel

A Appendix

AA Real-World Use Cases — Full Diagrams

The three diagrams discussed in Section 5.5.

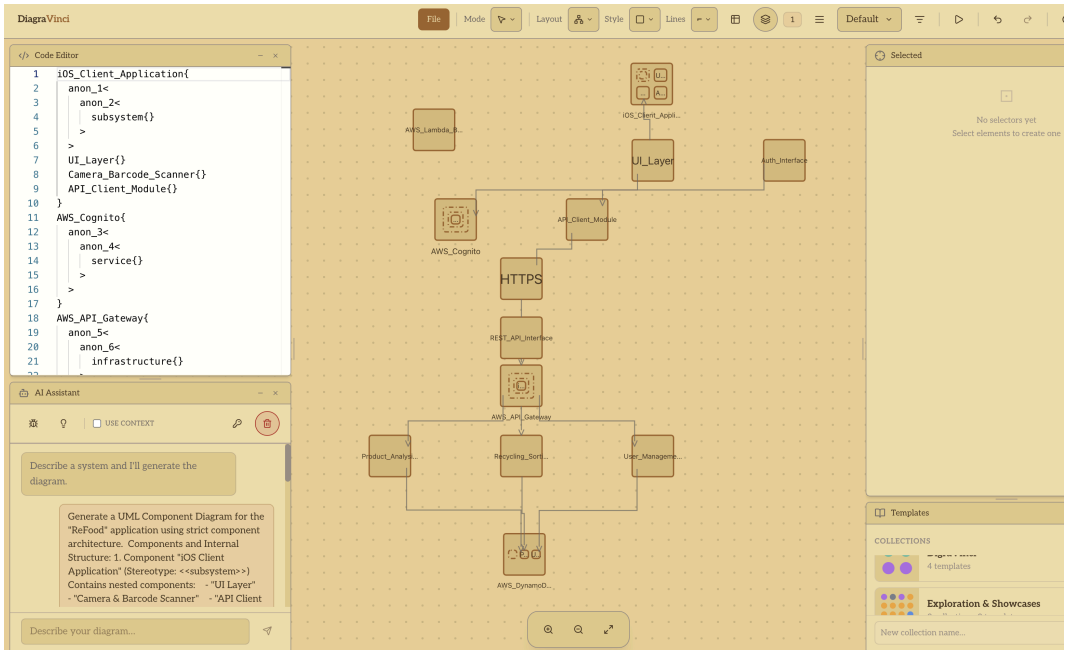


Figure 3: Recycling-app architecture — generated from an AI prompt (in-app view).

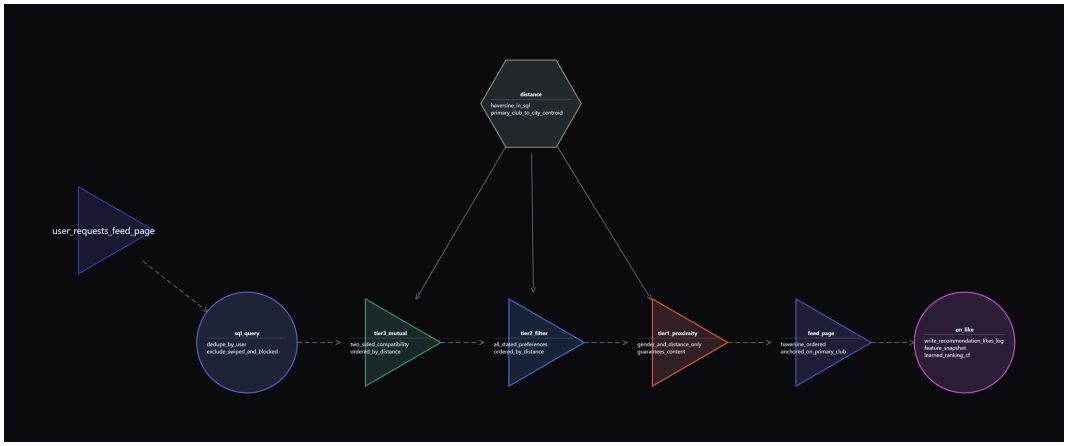


Figure 4: Feed-ranking pipeline, timeline layout: request, tiered filters, ranked feed.

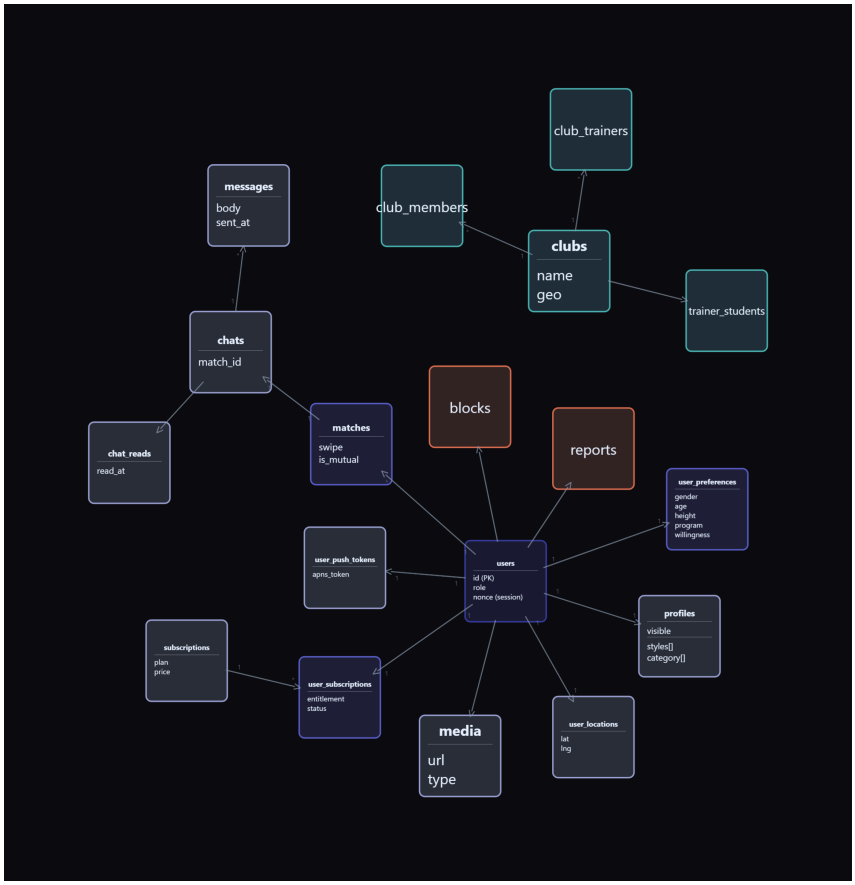


Figure 5: Fitness-dating data model: eighteen entities on one canvas.

AB Diagramming DiagraVinci Itself — Full Diagrams

Three diagrams of the tool, about the tool, referenced from Section 5.5.

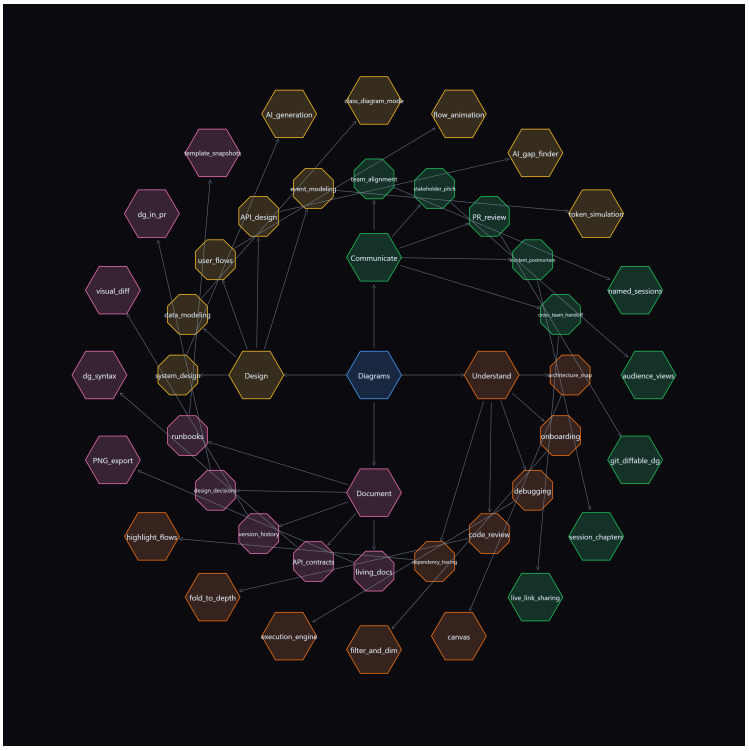


Figure 6: Use-case snowflake: what diagrams are for, organized around Design, Understand, Document, and Communicate.

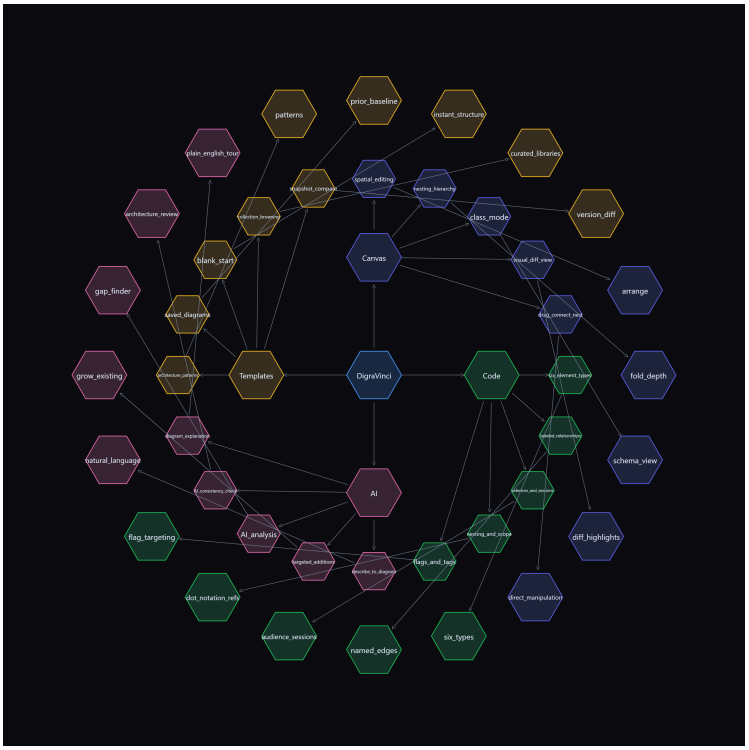


Figure 7: Feature snowflake: the four input modes — Templates, Canvas, Code, AI — and their sub-features.

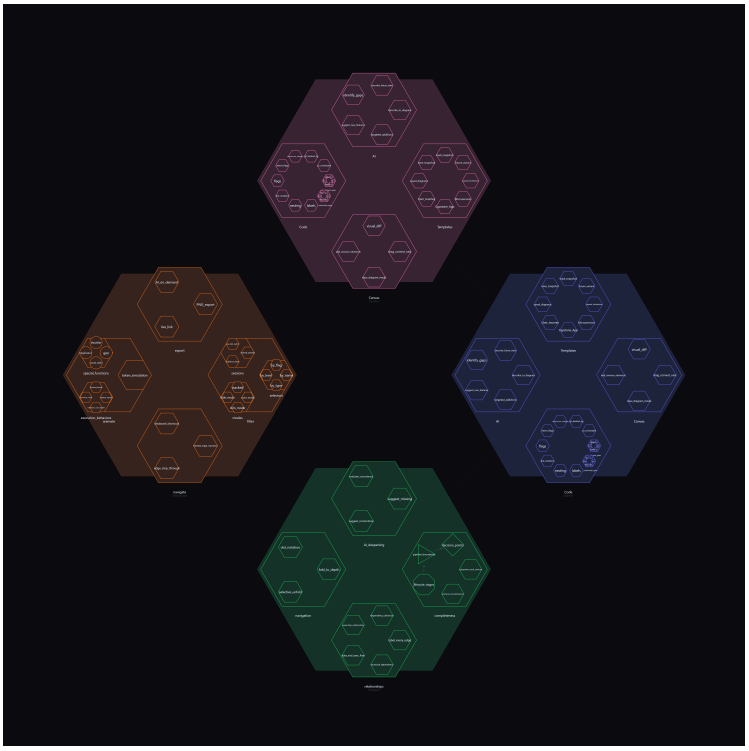


Figure 8: The Sketch–Deepen–Showcase–Evolve framework, each stage as its own cluster.

AC Codiagram: Tracing Flows Through Files

A per-file map of which user-facing flow touches which source file, and in what call order, referenced from Section 3.

Symbol	Meaning
`+`	The file backs this flow but is structural or shared support — always present, no specific position matters.
`-`	The file is not involved in this flow at all.
Whole number	A step where control crosses to a different actor that does not hand a value back into the caller's own continuing logic — a Redux dispatch, a callback fired into another subsystem, an independent loop starting. Increases caller to callee.
Floating-point	A synchronous call made by the file at the parent whole number, consumed by that same parent before it continues its own logic — nested, not handed off.
`a;b`	The file is reached from two distinct parent steps in the same flow, not sequentially.

Table 9: Codiagram notation legend.

Worked example, the code-edit flow (**`CD`**): typing in **`CodeEditor.tsx`** (**`1`**) hands off to **`SyncManager.ts`** (**`2`**), which synchronously tokenizes via **`Lexer.ts`** (**`2.1`**), parses via

Parser.ts (**2.2**), normalizes group and session directives via **groupUtils.ts** (**2.3**), diffs against the current model via **ModelDiffer.ts** (**2.4**), and merges view state via **ViewStateMerger.ts** (**2.5**) — all before **SyncManager** dispatches to **diagramSlice.ts** (**3**), a different actor, which notifies **VisualCanvas.tsx** (**4**).

Seven flows are tracked this way: code edits (**CD**), visual edits (**VIS**), UI interaction (**UI**), AI generation (**AI**), execution (**EXEC**), multi-tab sync (**TAB**), and the force-directed layout's own animation loop (**FORCE**), which runs alongside whichever of the first four triggered it. The full table, one row per file, lives alongside the codebase in **docs/architecture-codiagram.md**.