Capstone Project Report: Simulark

Intelligent Backend Architecture Design and Visual Simulation Platform

Project Title: Capstone Software Development Project

Module Name: CAI

Course Name: BDSE

Student Name: [Your Name]

Student ID: [Your ID]

Submission Date: January 24, 2026

1. Project Overview

Introduction

In the contemporary software engineering landscape, a critical dissonance exists between high-level system design and low-level implementation. Solutions Architects typically rely on static diagramming tools (e.g., Lucidchart, Draw.io) to visualize distributed systems. However, these artifacts are technically inert—they lack semantic awareness of the components they represent and become obsolete the moment implementation begins. Conversely, Infrastructure-as-Code (IaC) frameworks like Terraform are robust but lack the visual immediacy required for rapid prototyping and stakeholder communication.

Simulark addresses this architectural inefficiency. It is an AI-powered platform designed to bridge the "Design-to-Implementation Gap." By transforming natural language requirements into semantic, auto-arranged diagrams with "alive" visual data flows, Simulark acts as a high-fidelity Computer-Aided Design (CAD) tool for backend engineering. Crucially, it extends beyond visualization by acting as a **Context Bridge** for modern AI-assisted workflows, exporting architectural intent directly into the developer's Integrated Development Environment (IDE).

Objectives

The primary objectives of this capstone project are to:

1. **Engineer a High-Fidelity Interactive Canvas:** Develop a professional-grade visual editor using **React Flow** and **Shadcn UI** that supports custom, semantically rich node components (Gateways, Compute, Databases, Queues) capable of maintaining

referential integrity.
2. **Implement Dual-Agent AI Orchestration:** Leverage a multi-agent architecture (Planner + Renderer) to interpret natural language requirements and generate structured system graphs, mitigating the hallucination risks inherent in single-shot LLM prompts.
3. **Develop a Visual Protocol Simulation Engine:** Implement a simulation layer that visualizes data flow semantics—distinguishing between synchronous (HTTP/gRPC) and asynchronous (AMQP/Stream) protocols via distinct animation signatures—to enhance architectural legibility.
4. **Establish a Cross-Platform Context Bridge:** Differentiate Simulark from standard tools by implementing a robust export suite. This includes **Live Context URLs** for dynamic state sharing, and model-specific context files (gemini.md, claude.md, .cursorrules) to facilitate high-fidelity hand-off to AI coding assistants.
5. **Ensure Scalability & Performance:** Implement advanced rendering optimizations, including **Web Workers** for off-main-thread layout calculations, to ensure the canvas maintains 60 FPS performance during complex graph manipulations.
6. **Integrate Architectural Validation & Intelligence:** Incorporate a real-time validation engine to detect anti-patterns (e.g., circular dependencies, unshielded databases) and provide component-level **Cost Estimation**, ensuring designs are not just visually correct but operationally viable.

Scope

To ensure a robust and deliverable product within the capstone timeline, the project follows a strict Minimum Viable Product (MVP) scope focused on the Backend Engineering niche:

In Scope:

- **Visual Interface:** Custom React Flow implementation with specialized backend node types using **Tailwind v4**.
- **Onboarding Suite:**
  - **Template Gallery:** Pre-built, production-ready architectures (e-commerce, SaaS, IoT) to eliminate "Blank Canvas Paralysis."
  - **Interactive Tutorial:** A guided "first-diagram" experience.
  - **Example Prompts:** One-click starters (e.g., "Design a microservices app with auth and payments").
- **Architectural Intelligence:**
  - **Real-time Best Practice Alerts:** Detection of critical issues (e.g., "Gateway exposed without rate limiting," "Database has no backup queue," "Circular dependency detected").
  - **Cost Estimation:** Monthly AWS/GCP estimates displayed per node to drive enterprise awareness.
- **Visual Simulation:** Protocol-based animated edges (Standard vs. Async) and active node highlighting.
- **Auto-Layout:** Implementation of **Dagre** running in a **Web Worker** to automatically

arrange AI-generated nodes without freezing the UI.
- **Generative AI:** Text-to-Architecture pipeline using structured output validation (**Valibot**) and a **Dual-Agent Orchestration** strategy.
- **Context Bridge (Exports):**
  - **Live Context URL:** A secure, read-only JSON endpoint for real-time state consumption by IDEs.
  - **Model-Specific Markdown:** gemini.md and claude.md optimized for specific context windows.
  - **IDE Rules:** .cursorrules generation for Cursor/Windsurf.
  - **Visual Exports:** High-resolution PDF/PNG and Mermaid.js code copying.
- **Authentication:** Social Login (GitHub/Gmail) via Supabase OAuth2.
- **Documentation:** Automated API documentation generation using **Scalar**.
- **Iconography: Iconify API** for scalable, vendor-agnostic cloud logos.

Out of Scope:

- **Mathematical Stress Testing:** Complex throughput/latency prediction calculations (reserved for future "Pro" tiers).
- **Direct Code Generation:** Writing actual application source code (replaced by the high-fidelity Context Bridge).
- **Real-time Collaboration:** Multi-user live cursor tracking.

Methodology

The development followed a **Rapid Application Development (RAD)** methodology, compressed into an intensive 3-week execution plan. This approach prioritized "Building by Component," allowing for parallel development of the UI and the AI pipeline.

1. **Foundation (Week 1):** Setup of Next.js architecture, Supabase integration, and the React Flow Canvas.
2. **Intelligence (Week 2):** Implementation of the Dual-Agent AI stack (Solar/Mistral) and the Web Worker Layout Engine.
3. **Refinement (Week 3):** Visual Simulation, Context Bridge (Export) implementation, Validation Logic, and final polish.

2. Background and Problem Statement

Context and Motivation

Designing distributed systems—such as microservices or event-driven architectures—requires a deep understanding of component interactions. Developers often suffer from **"Blank Canvas Paralysis"**—the inability to start designing complex systems from scratch. Even with visual tools, they may create designs that are structurally unsound or cost-prohibitive. Furthermore, with the rise of AI Coding Assistants (Copilot, Cursor), developers face a new problem: **Context Loss**. When a developer moves from a diagram to an IDE, the AI assistant is unaware of the broader system architecture, leading to hallucinated imports or incorrect

service connections.

Problem Description

- **Semantic Disconnect:** Traditional diagrams are static pixels. They do not distinctively visualize the difference between a synchronous API call and an asynchronous queue message.
- **Context Hand-off Friction:** There is currently no standardized way to transfer the "mental model" of a system architecture into an AI coding assistant without manually typing lengthy system prompts.
- **Performance Degradation:** Web-based diagramming tools often suffer from UI freezes when arranging large graphs (nodes > 50) on the main thread.
- **Lack of Validation:** Standard tools allow users to create "illegal" architectures (e.g., circular dependencies) without warning.

Assumptions

- **Target Audience:** Backend/Full-stack developers using AI-assisted workflows who require a structured method to define and export architectural context.
- **AI Reliability:** The platform assumes that a dual-agent strategy significantly reduces individual model failure rates compared to single-model approaches.

3. Project Proposal and Planning

Timeline and Milestones

Sprint Phase	Dates	Focus	Deliverables
Sprint 1	**Jan 3 - Jan 9**	**Core Infrastructure & UI**	Next.js/Supabase Setup, React Flow Canvas, Custom Node Components (Tailwind v4), Iconify Integration.
Sprint 2	**Jan 10 - Jan 16**	**AI & Layout Logic**	Dual-Agent Integration (Solar/Mistral), Valibot Schema Validation, **Web Worker Layout Implementation**.

Sprint 3	**Jan 17 - Jan 23**	**Simulation & Exports**	Visual Flow Animation, **Validation/Cost Engine**, Context Bridge, Scalar Documentation, Final Testing.
Submission	**Jan 24**	**Final Release**	Project Submission, Final Report, Live Deployment.

Resource Allocation

- **Frontend Engineering (60%):** Extensive work on **React Flow** customization, canvas optimization (Web Workers), and **Tailwind v4** styling.
- **Backend & AI (30%):** Developing robust API routes, Supabase interactions, and tuning LLM system prompts.
- **UX & Integration (10%):** Designing the "Context Bridge" user flow, Onboarding Templates, and export formats.

4. System Design and Architecture

System Architecture Diagram

The platform utilizes a modern, type-safe stack centered around the T3 Stack principles (Next.js, TypeScript, Tailwind) extended with Supabase for data persistence.

```
graph TD
  User[User Browser]

  subgraph "Frontend Layer (Next.js 14+)"
    Canvas[Interactive Canvas<br/>(React Flow)]
    Worker[Web Worker<br/>(Layout Engine)]
    VisEngine[Visual Effects<br/>(CSS/SVG)]
    Validator[Validation Engine<br/>(Rules & Cost)]
    ContextBridge[Context Bridge Engine]
    UI[UI Components<br/>(Shadcn + Tailwind v4)]
  end

  subgraph "Server Layer (Next.js Server Actions)"
    AuthGuard[Auth Guard<br/>(Supabase SSR)]
    SchemaVal[Schema Validation<br/>(Valibot)]
```

```
    Orchestrator[Dual-Agent Orchestrator]
  end

  subgraph "AI Layer (OpenRouter)"
    Aggregator[Aggregator Agent<br/>(Solar Pro 3)]
    Generator[Generator Agent<br/>(Mistral Devstral)]
  end

  subgraph "Data & Auth Layer (Supabase)"
    Auth[OAuth2<br/>(GitHub/Gmail)]
    DB[(PostgreSQL)]
  end

  User --> Canvas
  Canvas -- Async Message --> Worker
  Canvas --> VisEngine
  Canvas --> Validator
  Canvas --> ContextBridge

  Canvas --> AuthGuard
  AuthGuard --> Orchestrator
  Orchestrator --> Aggregator
  Aggregator --> Orchestrator
  Orchestrator --> Generator
  Generator --> SchemaVal
  SchemaVal --> Canvas

  AuthGuard --> Auth
  AuthGuard --> DB
```

Architecture Description

1. **Frontend (The "Visual Engine"):** Built with **Next.js (App Router)** and **TypeScript**.
    - **Canvas Engine:** A highly customized React Flow instance. Nodes leverage **Iconify** for rich, vendor-agnostic logos.
    - **Web Worker Layout:** To prevent UI freezes during complex graph generation, the **Dagre** layout calculations are offloaded to a dedicated web worker (src/workers/layout.worker.ts).
    - **Validation Engine:** A client-side heuristic engine that runs in real-time to detect anti-patterns (e.g., circular dependencies) and calculate rough cost estimates based on node types.
    - **Single Source of Truth:** The **Architecture Graph JSON** is the authoritative state.

2. **Backend (The "Orchestrator"):** Leverages Next.js Server Actions to manage the AI pipeline.
    - **Validation: Valibot** validates LLM outputs to ensure they match the graph schema. Semantic validators check for referential integrity.
    - **Documentation: Scalar** automates API documentation.
3. **AI Dual-Agent Architecture:**
   Simulark employs a **"Plan-then-Render"** Multi-Agent Orchestration strategy to eliminate hallucinations and ensure structural validity. This adheres to the **Separation of Concerns** principle.
    - **Step 1: The Aggregator (Thinking Agent):**
        - **Model:** upstage/solar-pro-3:free (102B parameters).
        - **Role:** High-level system design reasoning.
    - **Step 2: The Generator (Coding Agent):**
        - **Model:** mistralai/devstral-2512:free.
        - **Role:** Takes the abstract plan and converts it into strict, validated JSON.
4. **Data & Auth: Supabase** handles authentication and persistence.

Data Model (Architecture Graph)

The core data structure is a strict JSON graph schema validated by Valibot.

```
classDiagram
  class Project {
    id: UUID
    user_id: UUID
    name: String
    nodes: Node[]
    edges: Edge[]
    provider: Enum (AWS, GCP, Azure, Generic)
  }
  class Node {
    id: String
    type: Enum
    data: NodeData
    position: Vector2
  }
  class NodeData {
    label: String
    serviceType: Enum (Compute, Database, Queue)
    validationStatus: Enum (Valid, Warning, Error)
    costEstimate: Float
  }
  class Edge {
    id: String
```

```
    source: String
    target: String
    protocol: Enum (HTTP, Queue, Stream)
}

Project *-- Node
Project *-- Edge
Node *-- NodeData
```

5. Data Preparation and Processing

Iconify & Tabler Hybrid Strategy

To ensure a professional and scalable interface, Simulark uses a hybrid icon strategy:

- **UI Icons (Tabler Icons):** Used for static interface controls.
- **Vendor Logos (Iconify API):** Used for dynamic node icons. This reduces bundle size while supporting an unlimited number of cloud providers via a cloud-based API.

Provider Switching Logic

The system maintains a mapping object. If the user switches the project provider to "AWS", a generic "Database" node automatically updates its icon to "RDS" or "DynamoDB" based on the mapping logic, without needing to regenerate the graph.

6. Application Development and Testing

Phase 1: Interactive Canvas & Performance (Development)

- **Custom Nodes:** Developed highly styled nodes using **Tailwind v4** and **Shadcn**.
- **Optimizations:** Implemented React.memo on custom node components to prevent unnecessary re-renders. Offloaded Dagre layout calculations to **Web Workers** (layout.worker.ts) to ensure the main thread remains unblocked during AI generation events.

Phase 2: Visual Simulation & Validation (Development)

- **Protocol Animation:** Implemented specific animation classes using CSS Keyframes for GPU-accelerated rendering (.animate-flow-fast, .animate-flow-slow).
- **Validation & Cost:** Implemented a real-time rules engine that flags best-practice violations (e.g., exposed Gateways) and aggregates monthly cost estimates per node to display a "Total Estimated Cost" badge.

Phase 3: AI & Context Bridge (Development)

- **Generation Pipeline:** Implemented the Dual-Agent flow using OpenRouter.

- **Context Bridge Engine:** Developed the logic to parse the internal graph state into multiple export formats:
    - **Gemini/Claude Parsers:** Specific string-builders that format the architecture into Markdown optimized for the specific context window tokens of different LLMs.
    - **Mermaid Transpiler:** A recursive function that traverses the graph to generate Mermaid graph TD syntax.

Testing Methodology

- **Unit Testing:** Validated the Dagre layout logic and Valibot schemas.
- **Visual Regression:** Checked that Tailwind v4 styles render correctly.
- **User Acceptance Testing (UAT):** Verified that the exported .cursorrules file correctly describes the architecture and provides meaningful context to external IDEs.

7. Visualization and Reporting

Intelligent Assistant Panel

A dedicated side panel displays the AI's breakdown of the architecture and acts as the control center for "Blueprint Generation" and **Real-time Alerts**. It lists active warnings (e.g., "Circular Dependency Detected") and provides a breakdown of the monthly cost estimate.

The "Context Bridge" Export Suite

Simulark differentiates itself through its comprehensive export capabilities, designed to act as the "source of truth" for external systems:

1. **Live Context URL:**
    - **Function:** Generates a secure, public read-only endpoint (e.g., https://simulark.app/api/context/uuid).
    - **Utility:** Allows AI Agents (like Cursor or Windsurf) to "read" the current state of the architecture via HTTP GET, ensuring the AI is always aware of the latest design changes without manual copy-pasting.
2. **Model-Specific Markdown (gemini.md / claude.md):**
    - **Function:** Exports a markdown description of the system optimized for specific LLMs.
    - **Utility:** claude.md utilizes XML tags (<context>...</context>) which Anthropic models prefer, while gemini.md uses structured hierarchical headers for Google models.
3. **Contextual Blueprints (.cursorrules):**
    - **Function:** Generates a ruleset file for the Cursor IDE.
    - **Utility:** Pre-prompts the IDE with the project's tech stack and architecture (e.g., "This project uses Next.js with Supabase. All database calls must be server-side.").
4. **Visual Exports (PDF / PNG):**
    - High-resolution snapshots for stakeholder presentations.
5. **Mermaid as Code:**
    - One-click copy of the diagram in Mermaid syntax for embedding in GitHub

README.md files.

8. Deployment and Integration

Deployment Architecture

- **Frontend & API:** Deployed on **Vercel** to utilize Edge Networks.
- **Database:** Hosted on **Supabase**.
- **Environment: T3 Env** enforces build-time environment variable validation.

System Integration

- **Auth:** Supabase Auth with OAuth2 (GitHub/Gmail).
- **Documentation: Scalar** for internal API reference.

9. Documentation and Presentation

Summary of Achievements

The project successfully delivers a high-end visual prototyping tool.

1. **Visual Excellence:** The use of **Tailwind v4** and **Iconify** creates a stunning, modern interface.
2. **Automated Layout: Dagre** integration via **Web Workers** solves the biggest pain point of generative diagrams (messy positioning & UI freezing).
3. **Architectural Maturity:** The **Context Bridge** transforms the tool from a simple whiteboard into a critical middleware in the AI-assisted coding workflow, while **Real-time Validation** ensures enterprise viability.

Lessons Learned

- **LLM Orchestration:** Splitting the AI task into "Reasoning" and "Generating" agents significantly reduced hallucinations compared to using a single model.
- **Performance at Scale:** Managing client-side layout for large graphs required careful memoization and state derivation to maintain 60 FPS.

Recommendations and Future Work

- **Advanced Layout:** Migrating from Dagre to **Elkjs** for better handling of nested sub-graphs (e.g., Services inside a VPC).
- **Round-trip Exports:** Enabling the import of Mermaid/JSON back into the tool to ensure sharing isn't a dead end.
- **Enterprise Features:** Introducing "Team Spaces" and audit logs for enterprise deployments.

10. References

1. Next.js Documentation. (2025). https://nextjs.org/docs

2. React Flow Documentation. (2025). https://reactflow.dev/
3. Supabase Documentation. (2025). https://supabase.com/docs
4. Valibot Documentation. (2025). https://valibot.dev/
5. Dagre Documentation. (2025). https://github.com/dagrejs/dagre
6. OpenRouter Documentation. (2025). https://openrouter.ai/docs
7. Iconify Documentation. (2025). https://iconify.design/docs/

11. Appendices

Appendix A: Valibot Schema for Architecture Nodes

The schema definition used to validate AI outputs and ensure strict type safety across the application.

{CODESNIPPET HERE}

Appendix B: Live Context Payload Example

An example of the JSON payload returned by the Live Context URL for IDE consumption.

{CODESNIPPET HERE}