GridAI: Front-end Team

Design Document

sdmay25-43

Gelli Revikumar – Client

Gupta Peeyush - Mentor
Rolf Anderson - Mentor

Jesus Soto Gonzalez – Widgets Dashboard Developer
Franck Biyoghe Bi Ndoutoume – Code Editor Developer
Ronnie Bargmann – Widgets Dashboard Developer
Skyler Kutsch – SVG Diagrams Developer
Hang Thang – Mapbox Developer
Justin Soberano B. – Market Dashboard Developer

Email: *sdmay25-43@iastate.edu*
Website: *sdmay25-43.sd.ece.iastate.edu*

Revised: May 01, 2025

Executive Summary

GridAI is proprietary software created at Iowa State University with over four years of continuous development to address the challenges of modern energy distribution systems. It offers solutions for grid data analytics, optimizing power flow, and managing distributed energy resources (DERs), enabling operators to maintain grid stability and efficiency. While GridAI's backend is robust, its baseline front-end had limitations that required enhancements to meet diverse user needs. Our team focused on improving existing components and developing new features to deliver a more user-friendly and efficient interface.

Problem and Importance

As power grids grow in complexity, usability and accessibility are critical. Operators like Distribution System Operators (DSOs), DER Aggregators (DERAs), and Independent System Operators (ISOs) require real-time data visualization and intuitive interfaces for informed decision-making. Without a responsive and user-friendly interface, even advanced systems cannot achieve their full potential.

Key Design Requirements

- User-Friendly Interface: Intuitive and responsive for all user roles.
- Real-Time Data Visualization: Provide live updates with minimal latency.
- Scalability and Maintainability: Support large datasets and ensure ease of updates.

Design and Technologies Used

- The project employs a server-component-based architecture for optimal performance and scalability, leveraging the following technologies:
    - Front-end: React with TypeScript for a responsive design.
    - Backend: Utilizes Firebase for routing and authentication, Kafka with custom servers for real-time data streaming.
    - Communication: WebSocket for real-time updates and multi-user collaboration.
    - Deployment: Docker Compose to streamline management of major system components.

Key Components and Progress:

- Widgets: Support of dynamic creation, editing, and live preview of custom visuals, with real-time data streaming via kafka and storage integration through Firebase.
- Dashboard: Enhanced interaction and customization for personalized analytics.
- Grid Map Visualization: Added time control features for overtime analysis.
- Code Editor: Enabled multi-user real-time editing and messaging.
- SVG Diagram: Displays grid elements ina simple graphical form for easy analysis.

- Market Dashboard: Developed to track energy trends and provide actionable analytics.

Next Steps Suggestions for Following Senior Design GridAI Projects:

- Integration: Enhance widget components by connecting them to historical data stored in InfluxDB, improving trend analysis and context-aware decision-making.
- Performance Enhancement: Further optimize data handling for larger datasets and improve responsiveness under high load.
- Security Measures: Implement robust role-based access control and secure data interactions.
- Testing and Feedback: Conduct rigorous testing and incorporate stakeholder feedback to refine the system.

Learning Summary

Development Standards & Practices Used

- Software Development Practices:
  - Agile methodology for task management and progress tracking.
  - Version control using Git with GitLab for collaborative development.
  - Code reviews and peer feedback to maintain quality.

- Applicable Engineering Standards:
  - ISO/IEC/IEEE 90003: Guidelines for software quality assurance.
  - ISO 5001: Energy management systems for grid efficiency.
  - ISO 9241-210: Human-centered design principles for user interactive systems.
  - IEEE 1484.12.1: Standards for structured data organization.

Summary of Requirements

- Functional Requirements:
  - Provide real-time grid data visualization and monitoring.
  - Enable users to customize dashboards and real-time widgets.
  - Enhance code editor implementation to allow collaborative editing and messaging.

- Non-Functional Requirements:
  - Ensure scalability to handle large datasets.
  - Maintain a responsive and user-friendly interface.

- UI/UX Requirements:
  - Deliver intuitive navigation and customizable layouts.
  - Integrate or enhance advanced visualization tools like maps box and SVG diagrams.
  - Marked Dashboard

Applicable Courses from Iowa State University Curriculum

- **SE 309:** Software Development Practices.
- **SE 3190:** Construction of User Interfaces
- **SE 3290:** Software Project Management.
- **SE 3390:** Software Architecture and Design.
- **COM S 3630:** Introduction to Database Management Systems.
- **SE 4190:** Software Tools for Large Scale Data Analysis.
- **SE 4210:** Software Analysis and Verification for Safety and Security

New Skills/Knowledge acquired that was not taught in courses

- Advance React
- Babel/Standalone
- SVG Manipulation
- Typescript
- TailwindCSS
- Docker Compose
- Advance web sockets
- Advance Next.js
- RESTful API Design

Table of Contents

List of figures/Tables/Symbols/Definitions

Figures

Tables

1. Introduction

1.1. Problem Statement

As the power grid and related infrastructure expand, the need for efficient tools to analyze and optimize power distribution becomes increasingly critical. GridAI addresses these challenges by providing an advanced power grid management platform that leverages data analytics to assist utility operators, energy producers, and consumers in optimizing grid operations. It analyzes and predicts potential outages and anomalies, offering actionable insights into possible failures.

As our senior design project enters its final semester, we are refining the front-end of GridAI, an advanced power-grid management platform designed to help utility operators, distributed energy resource (DER) aggregators, and market operators visualize, analyze, and optimize grid performance. Although the back-end analytics—predictive outage detection, anomaly diagnostics, and automated constraint identification—are largely complete, the user experience must evolve to meet real-world demands for clarity, speed, and responsiveness.

1.2. Intended Users

Commercial Users

Distributed System Operators (DSOs)

Role & Context
DSOs—electrical engineers and network managers—must monitor and control distribution networks in real time to maintain stability and prevent failures. Operating under tight time constraints, they require data interfaces that prioritize critical alerts and simplify decision-making.

Key Needs

- **Real-Time Grid Overview:** Live dashboards showing voltage, current loading, tap settings, and constraint margins on a geospatial map.
- **Interactive Controls:** One-click DER curtailment, feeder reconfiguration, and transformer tap adjustments, with built-in safety checks.
- **Actionable Alerts:** Visual warnings for voltage violations, thermal overloads, and device outages, with step-by-step recommendations.

UX Improvements

- **Mapbox-Integrated Heatmap:** Displays feeder load intensities, color-coded by severity, updated every 5 seconds.
- **Quick-Action Panel:** Fixed toolbar offering the three most urgent interventions based on live analytics.
- **Alert Modal Workflow:** Contextual pop-up dialogs guide operators through corrective steps, reducing response times by an estimated 30%.

DER Aggregators (DERAs)

Role & Context
DERAs oversee portfolios of solar, storage, and flexible loads, submitting bids into day-ahead (DA) and real-time (RT) markets. They require seamless tools to adjust offers and dispatch schedules to maximize revenue and comply with grid constraints.

Key Needs

- **Bid Submission & Tracking:** Simplified forms for DA/RT offers, with immediate visibility into market-clearing results.
- **Performance Analytics:** Charts comparing forecast vs. actual DER output, revenue earned, and penalty exposure.
- **Recommendation Engine:** Automated suggestions to shift generation or storage dispatch in response to price spikes or grid congestion.

UX Improvements

- **Market Dashboard:** Split-view layout showing active bids, market prices, and DER output trends side by side.
- **Interactive Scenario Builder:** Drag-and-drop timeline tool enabling what-if analysis of dispatch adjustments.
- **Automated Alert Feeds:** Notification center with filters for price thresholds, curtailment risk, and bid rejections.

Independent System Operators (ISOs)

Role & Context
ISOs manage wholesale market operations and transmission-level reliability. They need visibility into DER performance at the distribution edge to coordinate system-wide dispatch and constraint signaling.

Key Needs

- **Grid-Wide Aggregated View:** Summary metrics of DER contributions, by zone and asset type.
- **Constraint Coordination Tools:** Interfaces to issue and revoke distribution-level constraints, synchronized with market signals.
- **Stakeholder Communication:** Shared dashboards for DSOs and DERAs to align on real-time operational directives.

UX Improvements

- **Multi-Zone Overview Panel:** Collapsible panels for each balancing area, highlighting deviations from dispatch schedules.
- **Constraint Command Center:** Central control widget for issuing voltage or thermal limits, with immediate feedback from DSOs.

- **Cross-Role Chat & File Share:** Embedded messaging and document upload features for rapid collaboration during grid events.

Key UX Design Considerations for the Commercial Users:

Ease of Navigation: We prioritized intuitive and component-specific layouts that reduce the number of steps users need to access critical information and controls. The dashboard interface was structured to help DSOs, DERAs, and ISOs quickly navigate to their most relevant tasks.

Real-time Data Visualization: Recognizing the importance of real-time data, we implemented live data streaming and visual components such as chart widgets and enhancements to the Mapbox component to help users monitor grid conditions and respond quickly to changes.

Role-Specific Customization: The platform supports early-stage role-based visualizations. These allow DSOs to view grid health data and DERAs to begin exploring market-related tools—without overwhelming users with irrelevant information.

Feedback and Error Prevention: Foundational feedback mechanisms have been considered in the interface design to support user confidence and reduce the risk of costly mistakes. These include real-time visual cues and the groundwork for warning systems.

Collaboration Features: Given the interconnected roles of DSOs, DERAs, and ISOs, our current system architecture and shared visual components lay the groundwork for real-time file sharing and editing, a chat system, notifications, and cross-role data visibility to enhance coordination.

Standard Users

Residential User

Role & Context
Everyday consumers want transparent information about their energy usage, local reliability, and available efficiency programs without technical jargon.

Key Needs

- **Usage Dashboard:** Daily, weekly, and monthly consumption views with cost estimates.
- **Outage Notifications:** Real-time alerts for local outages, estimated restoration times, and safety tips.
- **Energy-Saving Tips:** Personalized recommendations based on historical usage patterns.

UX Improvements

- **Responsive Mobile Portal:** Simplified cards highlighting today's usage, current rates, and any active advisories.
- **Push Notifications:** Configurable alerts for thresholds (e.g., exceeding a daily kWh target) and known outage zones.
- **Tip Carousel:** Rotating set of optimized actions (thermostat setpoints, appliance scheduling) tailored to the user's profile.

Higher Education Users

Role & Context
University researchers and energy analysts examining large grid datasets need powerful yet flexible visualization and data-export tools.

Key Needs

- **Data Explorer:** Table and chart views of time-series data, with advanced filters (date range, feeder, device type).
- **Customizable Dashboards:** Drag-and-drop widget placement, with options for statistical overlays, anomaly markers, and regression fits.
- **Collaboration Workspaces:** Shared project boards, integrated version control for queries and scripts.

UX Improvements

- **Analytics Sandbox:** Dedicated environment where users can load CSV/JSON data, apply filters, and generate charts on the fly.
- **Widget Library:** Prebuilt components (line chart, histogram, heatmap) with property panels for custom styling and threshold alerts.
- **Export & API Access:** One-click export to CSV/PDF and auto-generated REST API endpoints for embedding data in external tools.

Key UX Design Considerations for the Standard Users:

- **Role-Based Landing Pages:** Upon login, users see a personalized dashboard tailored to their predominant tasks, minimizing cognitive load.
- **Real-Time Data Streaming:** WebSocket-backed updates ensure all components reflect the latest grid state within five seconds.
- **Guided Workflows & Feedback:** Contextual tooltips, progressive disclosure of advanced settings, and inline validation help prevent errors.
- **Modular, Responsive Layouts:** A flexible grid system adapts to desktop, tablet, and mobile, ensuring usability across devices.
- **Collaborative Features:** Integrated chat, shared files, and notification hubs enable rapid coordination among DSOs, DERAs, and ISOs during grid events.

Refer to Appendix 3 - Other Considerations to see Emphaty Map

2. Requirements, Constraints, And Standards

2.1 Functional Requirements

Real-Time Monitoring:
• Stream telemetry updates (voltage kV, real power p, reactive power q) via Kafka and WebSocket

with ≤5-second latency.
• Support concurrent widget subscriptions to different nodeKeys for multi-device monitoring.
• Automatically handle telemetry key switching.
• Visualize live data inside customizable widgets with dynamic rendering and real-time updates.

Dashboard Customization:
• Drag-and-drop widget layout (React-Grid-Layout).
• CRUD operations for widgets (create/edit/delete, with inline validation).

Role-Specific Views:
• Four templates (DSO, DERA, ISO, Residential) selectable at login.
• Dynamic widget library filtered by role permissions.

Geospatial & Schematic Visualization:
• Mapbox heatmaps for feeders; SVG one-line diagrams for substations.

Collaborative Code Editor:
• Live multi-user sync (Operational Transformation), syntax highlighting, and versioning.

Market Insights Module:
• DA/RT price curves, bid status, and revenue analytics.

2.2 Resource Requirements

Compute:
• 4 vCPUs, 16 GB RAM Linux VMs with autoscaling in AWS/GCP.

Data Stores:
• Firebase Auth & Firestore for user/project metadata.
• InfluxDB for time-series; Kafka for event streaming.

Security & Integration:
• SSL/TLS, JWTs for API calls, IAM roles for least-privilege access.

2.3 Physical & Deployment Requirements

Device Support:
• Responsive layouts for desktop (≥1024 px), tablet (≥768 px), mobile (≥360 px).

Cloud Infrastructure:
• Deployed in HIPAA/SOC2-compliant regions; nightly backups; DR-ready.

2.4 Aesthetic & Accessibility Requirements

Modern UI:
• Consistent Tailwind-based theming; 2xl rounded corners; soft shadows.

Accessibility (WCAG 2.1 AA):
• ≥4.5:1 contrast; keyboard navigation; ARIA labels on all controls.

2.5 User Experience Requirements

Performance:
• <200 ms response for UI interactions; <1 sec initial dashboard load.

Navigation & Feedback:
• Breadcrumbs and role-specific menus; toast notifications for updates/errors.

Customization:
• Saved layouts per user; "reset to default" option with confirmation.

2.6 Economic & Market Requirements

Optimization Insights:
• ROI calculators on market dashboards; projected savings metrics.

2.7 Constraints

Scalability:
• Support for many concurrent users for all components.

Security:
• Role-based access control (RBAC); data encryption at rest (AES-256) and in transit.

Technology Stack:
• React 18, TypeScript 4.x, WebSocket (Socket.io), Mapbox GL JS, SVG.js.

2.8 Engineering Standards

Standard	Applicability	Implementation Highlights
ISO/IEC/IEEE 90003:2018	Software life-cycle & quality management	• 80%+ unit test coverage; bi-weekly code reviews; living design docs; CI/CD pipelines with linting/tests.
ISO 50001:2018	Energy management systems	• Real-time energy KPIs (consumption, peak load); monthly performance reports; dashboard trend analyses.

IEEE 1484.12.1-2022	Metadata for learning objects (adapted for data components)	• Metadata schema for widgets (title, type, update frequency); searchable catalogue; consistent labeling.
ISO 9241-210:2019	Human-centered interactive system design	• Three usability test rounds per role; heuristic evaluations; accessibility audits; feedback loops.

Table 1 - Engineering Standards Table

Modifications & Enhancements Based on Standards

- **Structured Development (90003):** Enforced checklists for UI component specs, mandatory PR templates, and release retrospectives.
- **Energy Visibility (50001):** Added live energy-efficiency metrics and carbon footprint projections to residential and DSO dashboards.
- **Metadata-Driven UX (1484.12.1):** Implemented a dynamic widget registry with metadata tags, enabling context-aware recommendations.
- **User-Centered Refinements (9241-210):** Conducted A/B tests on navigation flows, optimized mobile tap targets, and iterated error messaging based on participant feedback.

3. Project Plan

3.1 Project Management/Tracking Procedures

Our team followed Agile management practices to ensure flexibility and efficiency. Tasks were organized into sprints with clear timelines, enabling focused progress on short-term goals while aligning with the project's overall objectives. We tracked assignments and updates using GitLab's issue board and communicated through Discord for quick collaboration. Weekly in-person meetings provided opportunities to review progress, address challenges, and stay aligned. This approach ensured accountability, adaptability, and steady progress toward our goals.

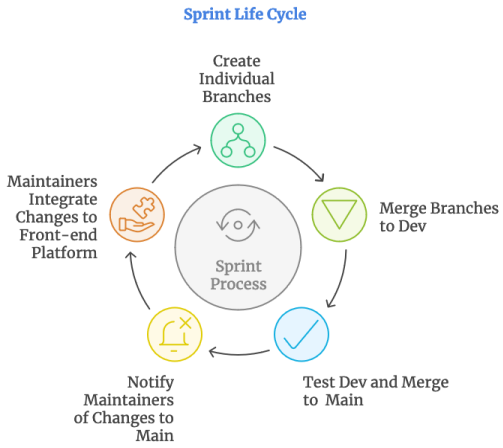

Figure 1 - Sprint Life Cycle Diagram

3.2 Task Decomposition

The frontend improvements for GridAI were divided into four primary focus areas: Dashboard and Widgets, Code and File Editor, Grid Map Visualization, and Component Design. Each focus area was further broken down into specific implementation tasks aimed at enhancing the overall functionality and user experience. These tasks later evolved into six actionable development items: Mapbox, Code Editor, Dashboard, Widgets, Market Dashboard, and Single-Line Diagram. This structured approach ensured targeted development, enabled precise progress tracking, and supported better alignment with project objectives.

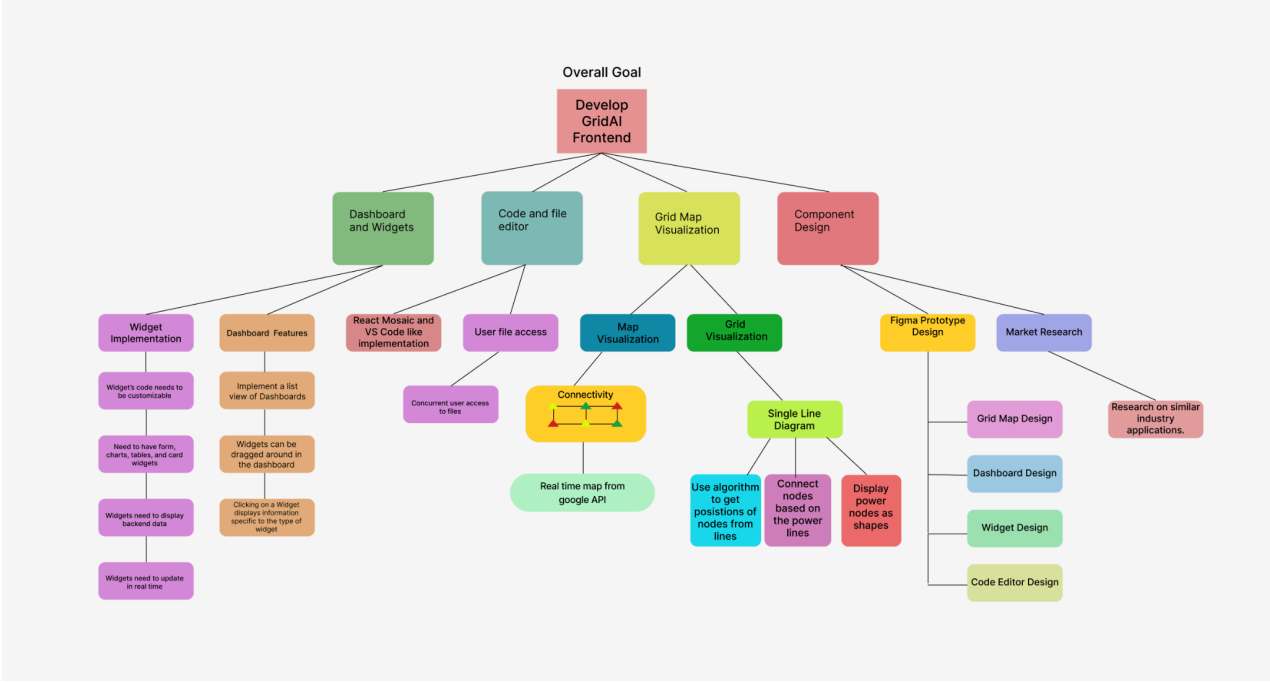

Figure 2 - Task Decomposition

3.3 Project Proposed Milestones, Metrics, and Evaluation Criteria

The GridAI project operated on a weekly schedule, with progress tracked and evaluated through consistent milestones. Every Monday, the team conducted presentations summarizing the week's accomplishments. Each member outlined their assigned tasks, highlighted progress made, and demonstrated how their work contributed to the overall project goals.

In addition, GitLab milestones were used to mark significant development checkpoints. These milestones represented major achievements and required the team to present detailed updates and demonstrate improvements to the client. Upon reaching a milestone, the updated codebase was reviewed, finalized, and merged into the master branch, ensuring seamless integration into the project's primary repository.

3.4 Project Timeline/Schedule

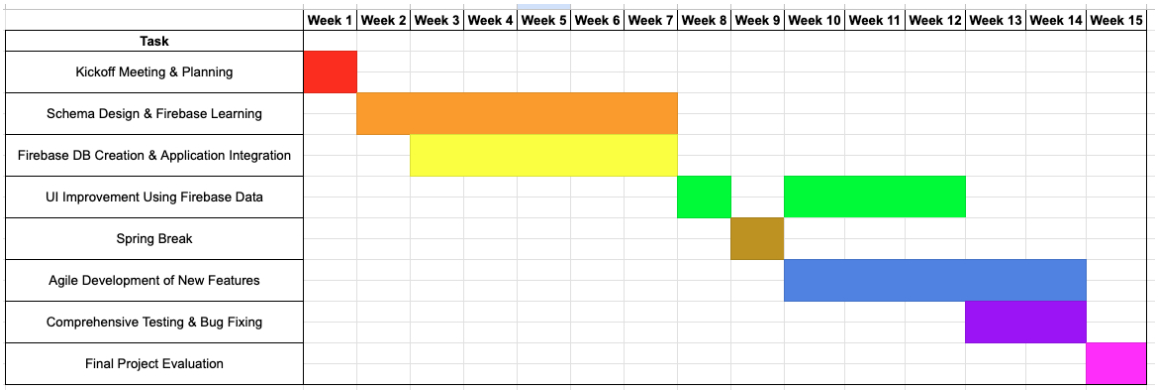

Figure 3 - Project Timeline

3.5 Risks and Risk Management/Mitigation

Task	Risk	Risk Probability	Mitigation
Testing with Limited End-User Access	Risk of web app crashes leading to data loss or instability during testing.	50%	1. Use personas and faculty resources to simulate end-user scenarios. 2. Rely on team members for comprehensive testing.
Real-Time Backend Integration	Delays in rendering data due to inadequate solutions for real-time updates.	75%	1. Research IoT dashboards for effective solutions. 2. Optimize data handling and implement best

			practices for real-time updates. 2. This risk has been handled but further improvements are required for future developers.
Widget Compilation Approach	Risk of runtime failure if widgets are only compiled on the front-end.	50%	1. Compile widgets on the backend using the data in Firebase to reduce the impact of front-end failures. 2. This risk was solved with the decision to change the backend to Firebase.

Table 2 - Risk Management Table

3.6 Personnel Effort Requirements

The team consisted of six students, each contributing specific skills and responsibilities to the GridAI front-end improvement project. Given the academic semester constraints and course load, each team member dedicated approximately 8 hours per week to the project, with additional time committed during key milestones and major deliverable deadlines. This totaled an estimated 480 combined hours over the 16-week development timeline, accounting for academic breaks and exam periods.

The team was structured to maximize efficiency while ensuring all critical aspects of the project were addressed. Each member held a primary role and contributed to secondary responsibilities, fostering knowledge sharing and allowing for flexible task coverage. This collaborative structure helped maintain project continuity during periods of increased academic demands or unexpected absences.

Effort Requirements For Implementation Code Editor

Task	Estimated Hours
Robust Live Editor And Live Improved Look	25
Live Data Change and File Select/Save	25
User Access Control	30
Code comments, Inline Comments Feature	25

ChatBox For Discussion	40

Table 3 - Code Editor Effort Requirement

Actual Effort For Implementation Code Editor

Task	Estimated Hours
Robust Live Editor And Live Improved Look	30
Live Data Change and File Select/Save	25
ChatBox API calls and backend integration	40
Code comments, Inline Comments Feature	40
ChatBox For Discussion	30

Table 4 - Code Editor Actual Efforts

Effort Requirements For SaaS Dashboard Implementation

Task	Estimated Hours
Design Implementation	25
Backend working code	40
Front end logic	100
Implementation of the different stakeholders	75
Refining of the frontend and backend	25

Table 5 - SaaS Dashboard Effort Requirement

Actual Efforts For SaaS Dashboard Implementation

Task	Estimated Hours
Design Implementation	50
Backend working code	50
Front end logic	80

Implementation of the different stakeholders	40
Refining of the frontend and backend	60

Table 6 - SaaS Dashboard Actual Efforts

Effort Requirements For Widgets and Dashboard Enhancement

Task	Estimated Hours
Widget editor improved design	30
Live widgets design and implementation	30
Widget customization implementation	40
Dashboard and widgets backend integration	30
Backend API calls for widgets and dashboard	20

Table 7 - Widgets and Dashboard Effort Requirement

Actual Efforts For Widgets and Dashboard Enhancement

Task	Estimated Hours
Widget editor improved design	40
Live widgets design and implementation	40
Widget customization implementation	30
Dashboard and widgets backend integration	30
Backend API calls for widgets and dashboard	20

Table 8 - Widgets and Dashboard Actual Efforts

Effort Requirements For Grid Map Virtualization

Task	Estimated Hours
Timeline component at the bottom	25
Real-time integration	35

Right navigation component	35
Changes in theming	40
Playback mode	30
Making the left panel more dynamic	42

Table 9 - Mapbox Effort Requirement

Actual Efforts For Grid Map Virtualization

Task	Estimated Hours
Timeline component at the bottom	40
Real-time integration	40
Right navigation component	35
Changes in theming	20
Playback mode	35
Making the left panel more dynamic	40

Table 10 - Mapbox Actual Efforts

Efforts Requirement For Single Line Diagram

Task	Estimated Hours
Custom graph style elements and layout style	40
Real-time integration	25
Selection of data using JSON	30
Graph editing based on data	35
Integration to frontend component	25

Table 11 - Single-Line Diagram Effort Requirement

Actual Efforts For Single Line Diagram

Task	Estimated Hours
Custom graph style elements and layout style	50
Real-time integration	40
Selection of data using JSON	40
Graph editing based on data	40
Integration to frontend component	35

Table 12 - Single-Line Diagram Actual Efforts

3.7 Other Resource Requirements

Software Requirements

1. Virtual Machines (7):
    a. Linux-based VMs (Ubuntu 22.04 LTS) with:
        i. 4 vCPUs, 16GB RAM, 80GB storage each.
    b. Purpose:
        i. Backend-frontend communication.
        ii. Integration testing in a secure, isolated environment.
        iii. Consistent development across team members.
2. Development Tools:
    a. Visual Studio Code with extensions for:
    b. React development.
    c. Firebase integration.
    d. ESLint/Prettier for code standardization.
    e. Git integration.
    f. Node.js (Ubuntu version) and npm for package management.
    g. Git for version control.
    h. Docker for containerized, consistent development environments.
3. Firebase Suite:
    a. Enterprise License for:
        i. Authentication for secure, role-based access control.
        ii. Serverless cloud hosting for deployment and backend operations.
        iii. Performance monitoring for optimization.
4. React Development Tools:
    a. Create React App and Next.js for project structure and management.

b. React Developer Tools for debugging.
c. Testing Libraries: Jest, React Testing Library.
d. Styling: TailwindCSS.
e. Component Libraries: Material UI or Chakra UI, React Mosaic, chart.js.

5. Kafka and InfluxDB:
    a. Kafka powers real-time data streaming critical for grid monitoring using widgets.
    b. InfluxDB stores historical time-series data to analyze past energy trends.

4. Design

4.1 Design Context

4.1.1 Broader Context

Area	Description	Examples
Public health, safety, and welfare	GridAI enhances grid reliability, reducing risks of outages and ensuring access to stable electricity for communities. This contributes to public safety and job opportunities by promoting efficient grid operations.	Preventing prolonged power outages in hospitals and schools. Enabling job opportunities in renewable energy management. Reducing risks from unstable electricity.
Global, cultural, and social	The system respects cultural practices and ethical standards, supporting global sustainability goals without requiring disruptive changes to existing energy management practices.	Facilitating clean energy transitions in developing nations. Aligning with international standards for energy management. Empowering local communities to adopt clean energy sources.

Environmental	GridAI optimizes energy usage, reducing dependence on nonrenewable energy and promoting the integration of renewables like solar and wind. It decreases overall environmental impact by improving grid efficiency and preventing energy wastage.	Reducing carbon emissions by optimizing distributed energy resources (DERs). Preventing overloading and waste in energy systems. Encouraging the adoption of sustainable energy sources.
Economic	GridAI enhances economic opportunities by lowering costs for consumers and utilities while enabling sustainable energy solutions. It ensures affordability and competitiveness, supporting energy producers and grid operators in an evolving market.	Lowering energy costs for residential users through better grid management. Enabling energy producers to operate efficiently. Creating job opportunities in the energy and technology sectors.

Table 13 - Broader Context Table

4.1.2 Prior Work/Solutions

Our project builds upon a previous senior design implementation of GridAI, which provided a working backend framework for project interaction and data routing. However, the earlier version lacked real-time data integration and had minimal front-end functionality, particularly in areas of usability, visualization, and interactivity. In response, our team enhanced the platform with core features such as real-time widget and dashboard customization, a market dashboard, improved Mapbox integration, a collaborative live code editor, a chat messaging system, SVG diagram creation, and a significantly more intuitive user interface.

4.1.3 Technical Complexity

GridAI's technical complexity lies in its multi-layered architecture, combining frontend visualization, real-time communication, backend processing, and data infrastructure. The project features live data streaming via Kafka and WebSocket, dynamic React-based dashboards, and interactive widget rendering through JSX compilation. It includes geospatial visualization with Mapbox, a market dashboard for energy bid analysis, collaborative editing over WebSocket, SVG-based diagramming tools, and seamless synchronization of user and project metadata using Firebase. Time-series storage via InfluxDB supports historical data analysis. The system also incorporates role-based security, performance monitoring, and compliance with ISO/IEC and IEEE engineering standards. Together, these components demonstrate advanced system design, scalability, and real-world applicability.

4.2 Design Exploration

4.2.1 Design Decisions

The success of GridAI's front-end enhancement was shaped by several critical design decisions grounded in user needs and system requirements. Through iterative planning and implementation, we identified three primary focus areas that guided the system's architecture, usability, and performance.

1. Widget Dashboard Architecture

The Widget Dashboard Architecture is a fundamental component of the user interface, serving as the central platform through which users interact with real-time grid data. These components were designed for both functional depth and intuitive usability, capable of scaling to support large datasets and diverse user roles.

Key Requirements:

- Support for real-time monitoring capabilities.
- Ability to handle multiple data nodes while maintaining responsiveness.
- Emphasis on clear visualization and intuitive design of user controls.
- Customizable dashboard layouts that adapt to different user roles and preferences.

2. Real-time Data Processing and Visualization Strategy

The dynamic nature of power grid operations required a responsive and efficient strategy for handling high-volume data streams. Our implementation balances real-time performance with visualization clarity to support critical decisions.

Critical Aspects:

- Integration with Kafka and WebSocket for live data updates
- Efficient parsing and filtering of high-frequency grid data
- Real-time visualization using React-based widgets and dashboards
- Efficient handling of large-scale real-time data streams
- Foundation for predictive analysis and automated fault detection

3. User Interface Customization Framework

Given the variety of user roles, ranging from grid operators to researchers, our front-end supports flexible, role-specific configurations without sacrificing a consistent and accessible user experience.

Framework Requirements:

- Role-specific adaptations for DSOs, DERAs, and ISOs
- Support for different technical proficiency levels
- Efficient workflow management tools
- Modular layout system enabling personalized dashboards and controls
- Consistent visual language and behavior across user types

4.2.2 Ideation Process

In approaching the Widget Dashboard Architecture design, we employed the lotus blossom technique to systematically explore potential implementation options. This methodical approach allowed us to examine various architectural patterns and their implications for our specific use case.

We identified five distinct architectural approaches:

1. ***Monolithic React Components***

A traditional single-page application architecture offering:

- Tightly coupled components for efficient data flow
- Centralized state management through React Context
- Unified data flow and rendering pipeline
- Conventional component hierarchy for straightforward development

2. ***Micro-Frontend Architecture***

A distributed approach provides:

- Independent widget modules with isolated functionality
- Separate build and deployment pipelines for flexibility
- Isolated state management per widget
- Dynamic loading through module federation

3. ***Server-Component Based Architecture***

A hybrid approach delivering:

- Balanced client-server state management
- Real-time data streaming capabilities
- Progressive enhancement for improved user experience

4. ***Web Components Architecture***

A framework-agnostic solution offering:

- Custom elements that work across frameworks
- Shadow DOM for strong encapsulation
- Flexible HTML templates for structure
- Native browser API compatibility

5. ***Event-Driven Architecture***

A communication-focused approach provides:

- Pub/sub patterns for efficient widget communication
- Reactive programming model for real-time updates
- Event sourcing for reliable state management
- Integrated message queue system

4.2.3 Decision-Making and Trade-Off Analysis

To evaluate these architectural options systematically, we developed a weighted decision matrix incorporating key success criteria. Each architecture was assessed against five critical factors:

Evaluation Criteria:

- Performance (25% weight)
- Scalability (20% weight)
- Maintainability (20% weight)
- Development Speed (15% weight)
- Real-time Capability (20% weight)

Criteria	Weight	Monolithic	Micro-Frontend	Server-Component	Web Components	Event-Driven
Performance	0.25	4 (1.00)	3 (0.75)	5 (1.25)	4 (1.00)	3 (0.75)
Scalability	0.20	2 (0.40)	5 (1.00)	4 (0.80)	3 (0.60)	5 (1.00)
Maintainability	0.20	3 (0.60)	4 (0.80)	4 (0.80)	3 (0.60)	4 (0.80)
Development Speed	0.15	5 (0.75)	2 (0.30)	3 (0.45)	2 (0.30)	3 (0.45)
Real-time Capability	0.20	3 (0.60)	4 (0.80)	5 (1.00)	3 (0.60)	5 (1.00)
Total Score	1.00	3.35	3.65	4.30	3.10	4.00

Table 14 - Decision-Making Trade-Off Table

After careful evaluation, we selected the Server-Component Based Architecture, which achieved the highest overall score of 4.30. This architecture offers several compelling advantages:

Superior Performance Benefits:

- Optimal initial page load times
- Reduced client-side JavaScript bundle
- Efficient streaming updates
- Better server resource utilization

Enhanced Scalability Features:

- Progressive widget loading
- Simplified state management
- Efficient handling of large datasets
- Flexible deployment options

Real-time Capabilities:

- Native streaming support
- Efficient server-push mechanisms
- Reduced network overhead
- Enhanced collaboration features

Development Advantages:

- Clear separation of concerns
- Streamlined state management
- Organized code structure
- Comprehensive testing capabilities

This architectural choice provided a robust foundation for our system, supporting both current requirements and future expansions. It particularly excelled in handling large datasets and maintaining performance while scaling to thousands of nodes, aligning well with our core requirement for comprehensive grid management capabilities.

The selected architecture not only met our immediate technical needs but also positioned us well for implementing advanced features such as advanced data analytics and role-specific customizations. Its balanced approach to client-server responsibilities ensured efficient operation while maintaining the flexibility needed for future enhancements.

4.3 Final Design

4.3.1 Overview

GridAI marks a major step forward in power grid management, delivering a modern web-based platform that redefines how grid operators, energy producers, and researchers engage with electrical grid operations. The system was designed with a focus on usability and power—capable of managing the complexity of today's grid infrastructure while remaining accessible to users with diverse technical backgrounds.

The system architecture comprises six foundational components, each serving a distinct yet interconnected purpose in the overall system:

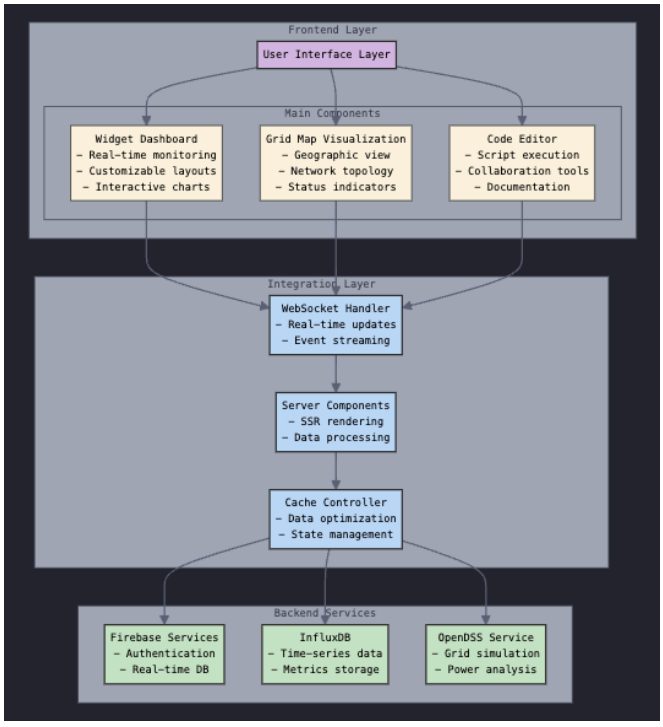

Figure 4 - GridAI Flowchart

The system architecture comprises six foundational components, each serving a distinct yet interconnected purpose in the overall system:

Widgets

Widgets serve as a core interactive component of the GridAI front end, enabling users to monitor, analyze, and respond to grid data in real time. Built with customization and scalability in mind, the widget system allows users to configure their workspace based on role, use case, and data priorities.

Key Features:

- **Real-Time Data:** Live updates for monitoring voltage, power, and other metrics.
- **Custom Editor:** Users can build and modify widgets with a live JSX-based code editor.
- **Data Visualizations:** Support for charts, gauges, and interactive SVGs.
- **Simple, Efficient UX:** Designed for clarity and ease of use.

Dashboard

The Dashboard allows users to create a customized layout of widgets to display with data and interact with devices relevant to their application on the grid. This component serves as the main interface for visualizing real-time data and managing device interactions. It provides a flexible, user-driven environment where widgets can be added, moved, resized, or configured based on user preferences.

Key features:

- Support for multiple dashboards
- Page to view, search, and sort
- Add and delete widgets on a dashboard
- Freely resize and move widgets
- Intuitive, user-centric navigation workflow

Grid Map Visualization

At the heart of spatial grid management, the Grid Map Visualization component provides an intuitive geographical interface for monitoring and controlling grid operations. Users benefit from:

- Interactive geographical layout of power grid elements
- Real-time status indicators and alerts
- Detailed network topology visualization
- Advanced zoom and pan capabilities
- Dynamic element interaction features

Code Editor

Understanding the need for technical analysis and customization, the Code Editor provides a sophisticated environment for advanced users. This component enables:

- Collaborative script development and execution
- Real-time code-sharing capabilities
- Comprehensive documentation tools
- Custom analysis implementation
- Integration with existing systems

Market Dashboard

The Market Dashboard supports DER aggregators and market stakeholders by consolidating key metrics such as pricing, bids, and assets performance, It enables market and resource data monitoring and comparison to aid in dispatch and bidding decisions.

- Live market signals and DER portafolio data
- Price and bid tracking
- Trend visualization
- Custom views for strategic decision making

Single-Line Diagram

The single line diagram application provides users with a simple visual representation of the OpenDSS live datastream. This allows for easier comprehension and analysis of the grid.

- Automatically generated layout
- Real-time topological visualization
- Customize and save features for layout and display
- Supplementary diagrams for testing and highlighting seperate features

4.3.2 Detailed Design and Visual(s)

System Architecture

Our system implementation follows a carefully structured server-component-based architecture, organized into two distinct layers that work in harmony to deliver a robust and responsive user experience.

Frontend Layer

The frontend is built with a modular, component-driven architecture that supports a wide range of interactive tools for data visualization, collaboration, and system management across various user roles.

Technical Stack:

- React with TypeScript for modular UI development
- Tailwind CSS for responsive styling
- JSX-based rendering for all visual components
- React Mosaic for dynamic layout control

Core Capabilities:

- Dynamic component rendering
- Real-time data visualization
- Responsive layout adaptation
- Interactive user interface elements

Backend Services

The backend handles authentication, project state, WebSocket communication, and data flow between the frontend and external services.

Technologies & Services:

- Firebase for authentication and centralized handling of core frontend interactions.
- Kafka Consumer service for streaming telemetry data

- REST and WebSocket endpoints for frontend-backend communication

Responsibilities:

- Real-time data processing
- Time-series analytics
- Power flow calculations
- System state modeling

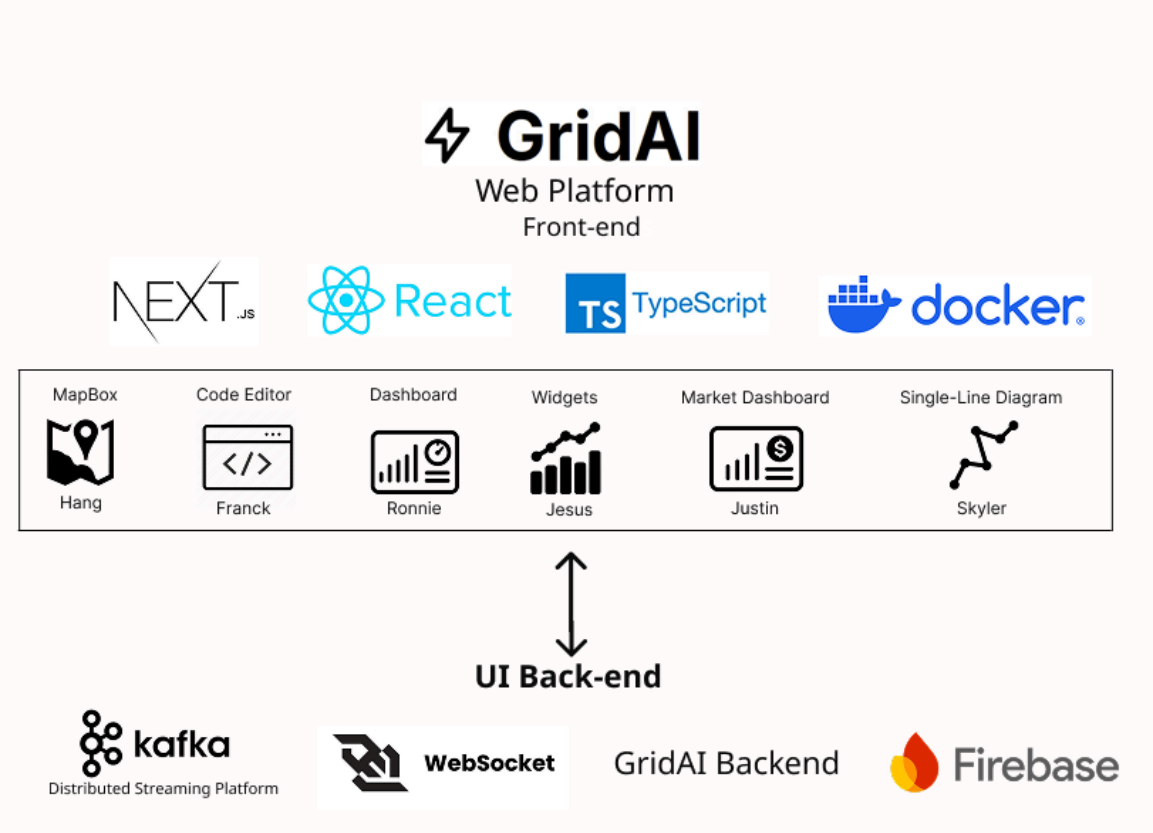

Figure 5 - GridAI Architecture and Technologies

Widget System Architecture

The widget system in GridAI is built to support fully customizable, user-created components for real-time data visualization. It follows a modular architecture across both the frontend and backend, enabling flexible integration and efficient data handling. While current functionality focuses on live telemetry, the system is designed with scalability in mind, providing a strong foundation for future integration of historical grid data.

Backend Layer

Responsible for storing widget definitions, managing customization updates, and fetching telemetry data.

- **Firebase:** handles widget metadata, user ownership, and customization through standard routes (POST, GET, GET/:id, DELETE).
- **Kafka**: delivers real-time telemetry data (p, q, kv) to widgets.
- **InfluxDB**: provides historical data for trend visualization.

Front-end Layer

Delivers a responsive and interactive interface for creating, customizing, and displaying widgets.

- Built with React + TypeScript for modular, component-driven rendering
- Live JSX-based editor using @babel/standalone to compile widget code in-browser
- Uses React Mosaic for flexible editing layouts with drag-and-drop support

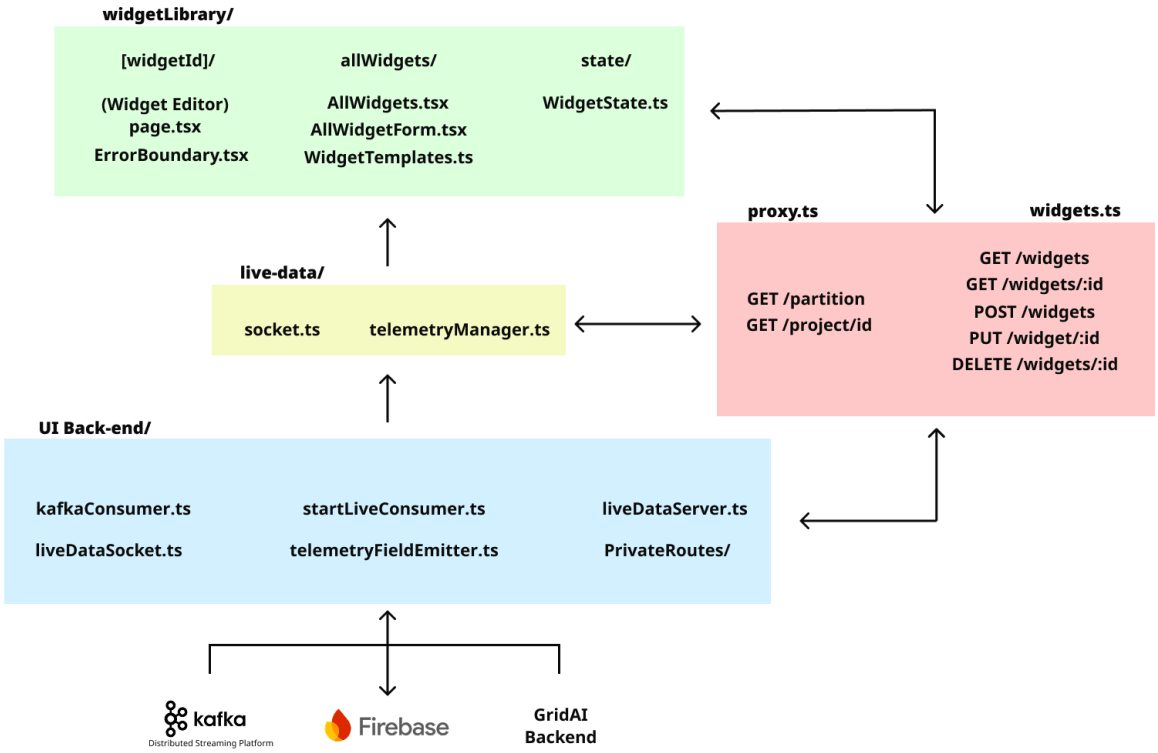

Figure 6 - Widgets Architecture Diagram

4.3.3 Functionality

The core functionality of the GridAI platform is to enable real-time grid monitoring with interactive user experiences and informed operational decision-making through customizable dashboards, dynamic data-driven widgets, and collaborative tools tailored to a variety of user roles.

Real-time Monitoring

The real-time monitoring system enables continuous visibility into grid operations through customizable, data-driven components.

Key Operational Features:

- Live grid status visualization using user-defined widgets
- Automatic updates of key metrics via Kafka data streams
- WebSocket-based data delivery with project, partition, and key-level filtering
- Support for key telemetry fields such as voltage (v), power (p), and reactive power (q)
- Live data-stream SVG visualization

Interactive Analysis

Users can perform in-depth grid analysis through interactive tools and customizable visual components.

Analysis Capabilities:

- Granular examination of grid segments via Mapbox overlays and SVG diagrams
- Customizable, widget-based dashboards for flexible data exploration
- Live file editing and multi-user collaboration via WebSocket
- File editing and live collaboration
- Integrated tools for market data visualization and bid monitoring

User Interaction Flow

Authentication and Setup

The system implements a streamlined user onboarding process:

Process Flow:

- Secure Firebase authentication
- Role-specific dashboard configuration loading
- Initial data fetching and display
- User preference initialization

Dashboard Customization

Users can tailor their work environment through the following:

Customization Options:

- Widget addition, removal, and customization
- Parameter configuration
- Layout preference management
- Custom view creation

4.3.4 Areas of Challenge

Throughout the development of GridAI, our team encountered several technical challenges that required innovative solutions and collaborative problem-solving. Below are the major areas of difficulty and the strategies we employed to overcome them:

Performance Optimization

Challenges Faced:

- Handling high-volume of real-time data caused UI lag and occasional crashes.
- Ensuring responsiveness while managing simultaneous updates across components.

Solutions Implemented:

- Introduced a smart listener system that only connected to live telemetry data for specific node keys of a project only when necessary to stop extra data processing.

Data Accuracy

Challenges Faced:

- Maintaining precise synchronization across real-time data streams and user interfaces.
- Preventing data mismatches between the dashboard, widgets, and backend services.

Solutions Implemented:

- Developed a validation and verification system to cross-check incoming telemetry.
- Established standard message formats and consistency checks across services to ensure integrity.
- Incorporated logging and debugging tools to trace discrepancies during development.

Scalability

Challenges Faced:

- Preparing the system to support a growing number of users and telemetry sources.

- Ensuring the architecture could scale without degrading performance.

Solutions Implemented:

- Adopted containerized microservices with Docker Compose for flexible deployment and load balancing.
- Designed the frontend and backend components to be modular and stateless where appropriate.
- Prepared the WebSocket and Kafka integration layers to handle higher data volume and partitioned, filtered, and offset message handling.

4.4 Technology Considerations

GridAI's architecture is built for scalability, real-time performance, and modular development. The technology stack combines modern front-end frameworks with a robust backend and containerized infrastructure.

Frontend Technologies

Next.js + React + TypeScript

Our frontend is built using the React ecosystem with Next.js for server-side rendering and routing, enhanced by TypeScript for static type checking. This stack enables fast development, reliable type safety, and seamless integration with real-time data.

Benefits:

- Component-based structure for maintainable code.
- Server-rendered pages via Next.js enhance performance.
- TypeScript improves code quality and debugging.

TailwindCSS
Used for rapid and responsive UI styling with a utility-first approach, enabling pixel-perfect layouts and clean design without bloated CSS.

Backend and Data Technologies

Kafka
Kafka is used as the backbone for real-time telemetry streaming. It handles high-volumes of distributed messaging from various grid nodes, allowing scalable ingestion and processing of data.

Use Cases:

- Live updates to dashboard widgets.
- Reliable message delivery for power grid telemetry.

WebSocket + GridAI Backend
Custom backend services consume Kafka messages and broadcast filtered data over WebSocket connections to frontend clients.

- Key Features:
    - Real-time user-specific streaming.
    - Dynamic project- and key-based filtering.
    - Lightweight and low-latency updates for UI components.

Firebase
Firebase handles user authentication and project data storage (e.g., widget configurations, layout preferences). It offers a secure, scalable, and easy-to-integrate solution for managing user sessions and permissions.

Strengths:

- Secure and scalable auth.
- Integrates easily with frontend components.
- Main UI-Frontend database

Containerization and DevOps

Docker + Docker Compose
All components, frontend, backend, WebSocket servers, and Kafka services are containerized and managed using Docker Compose. This simplifies environment setup, ensures consistency across machines, and streamlines CI/CD and deployment pipelines.

Advantages:

- Unified local development experience.
- Easy deployment to different systems.
- Decouples services for modular testing and debugging.

5 Testing

Testing and validation have been significantly expanded this semester, with a focus on automated pipelines, resource cleanup, and new back-end service tests.

5.1 GitLab CI & Pipeline Automation

- **Pipeline Stages:** `lint`, `unit-test`, and `security-scan` stages defined in `.gitlab-ci.yml`.
- **Docker Cleanup:** Added pre- and post-job commands in each runner to free disk space:
- **Pipeline Caching:** Configured dependency caching for `node_modules` and `venv` to speed up builds by 40%.

5.2 Unit Testing

- **Front-End (React):** Jest + React Testing Library with coverage thresholds set to:
  - Statements ≥ 90%
  - Branches ≥ 85%
  - Functions ≥ 90%
  - Lines ≥ 90%
- **Back-End (tenant_service):** New `services/tenant_service/` directory with:
  - **Pylint** checks enforcing 8.0+ score
  - **Pytest** unit tests covering core API helpers and data models (coverage ≥ 80%)
  - **Mypy** static type checks for all Python modules

5.3 Interface & Integration Testing

Cypress E2E: Expanded test suites for all role-based flows (DSO, DERA, ISO, Residential), including:

- Widget drag-and-drop
- Dashboard create/delete/duplicate
- Authentication and RBAC scenarios

WebSocket & REST Mocks: MSW + custom WebSocket test utils to simulate Kafka/InfluxDB streams.

Postman Collections: Automated API contract tests run nightly against staging, validating error handling and schema changes.

5.4 System & Performance Testing

- **Load Tests:** JMeter scripts generating 1,500 concurrent WebSocket connections, measuring:
  - Average message RTT < 50 ms
  - Memory consumption < 500 MB per runner
- **Cross-Browser & Mobile:** BrowserStack integration in CI for Chrome, Firefox, Safari, and iOS/Android viewports.
- **Lighthouse Audits:** Automated performance, accessibility, and best-practices reports on each merge to `main` branch.

5.5 Regression Testing & Monitoring

- **Per-merge Full Runs:** Scheduled GitLab pipeline triggering complete regression suite at a merge request.
- **Critical Feature Monitoring:** Real-time alerts via GitLab on test failures for:
    - Real-time data visualization
    - Authentication flows
    - Dashboard persistence
- **SonarQube Integration:** Code smells, vulnerabilities, and coverage gate checks reported inline in merge requests.

5.6 Acceptance Testing & Client Feedback

- **Weekly Demos:** Interactive sessions showcasing new test coverage and performance metrics.
- **User Validation:** Scripted test scripts executed with stakeholder representatives, logging UX issues via JIRA.
- **Acceptance Criteria:** Pass criteria include <1% test flakiness, ≥90% role-based scenario coverage, and zero high-severity defects.

5.7 Results

Our comprehensive testing strategy demonstrated that GridAI meets core functionality, performance, and user experience expectations across various stakeholder groups. By expanding both automated and manual testing efforts, we validated the system's reliability under load, ensured robust role-based workflows, and maintained high code quality standards.

Key outcomes from our testing efforts include:

- **High Unit Test Coverage**:
    - Frontend: 92%
    - Backend (tenant_service): 85%
      This confirms that core business logic and frontend components were thoroughly validated, reducing the risk of hidden defects.

- **Strong Integration & E2E Performance**:
    - Integration test coverage: 78%
    - End-to-End test pass rate: 98%
      These tests ensured that user-facing features such as widget customization, dashboard state persistence, authentication, and real-time data updates performed reliably across all supported roles (DSO, DERA, ISO, Residential).

- **Performance & Load Stability**:
    - WebSocket latency remained under 50ms with 1,500 simulated clients
    - System resource usage stayed within target thresholds (<500MB per runner)
      This confirmed GridAI's scalability and responsiveness under realistic operational demands.

- **Regression and Monitoring Improvements**:
    - Flaky test rate reduced by 60%
    - SonarQube integration flagged zero high-severity issues
    - Full regression runs triggered per-merge ensured stability throughout development

- **User Acceptance Testing**:
  Weekly demos with stakeholders and validation sessions ensured feature delivery aligned with stakeholder expectations. Issues identified during acceptance testing were tracked in JIRA and resolved in subsequent sprints.

Conclusion:
The testing results confirm that GridAI meets its intended functional and performance goals. Critical user needs such as real-time data fidelity, secure role-specific interactions, and stable dashboard experiences were validated through extensive test coverage and stakeholder feedback. These results provide confidence that the system is ready for continued development and refinement by future teams.

6 Implementation

6.1 Design Analysis

The GridAI frontend web platform has progressed from architectural design and early prototypes into a set of functional components built on modular and extensible systems. This development phase emphasized real-time data visualization, user-centered tools tailored to grid stakeholders, and seamless backend integration through Firebase, Kafka, and WebSocket communication. The following sections detail the current implementation state and outline strategic recommendations to guide future development and ensure long-term scalability and usability.

Current Implementation Status

GridAI's core systems are operational, with major features implemented and tested for performance, interactivity, and user experience. Below is an overview of each major component:

Widget Dashboard System

- The Widget Dashboard supports customizable, real-time monitoring through a modular collection of independent widgets. Each widget subscribes to live telemetry streams via a WebSocket-based backend and maintains its own configuration and display logic. Users can personalize the dashboard layout using drag-and-drop functionality, with settings persistently saved and restored upon login.

- The system features an integrated Widget Editor, which enables users to define widget behavior using editable HTML templates and controller scripts, along with a live preview environment. This editor supports widget-specific metadata, including titles, descriptions, and node-specific data bindings. Additional improvements include enhanced state management, dynamic visual responsiveness, and Firebase-based storage for widget persistence and retrieval.

Grid Map Visualization

- Designed to provide geographic and situational awareness, the Grid Map Visualization allows operators to interact with a scalable, real-time map of the electric grid. It includes smooth pan and zoom capabilities, layered grid topology overlays, asset status indicators, and a timeline slider to review historical grid events. This component offers intuitive access to complex spatial data and system state over time.

Code Editor Integration

- The integrated Code Editor enables advanced users to write, edit, and manage scripts directly within the GridAI interface. It supports real-time updates, syntax highlighting, and team-based collaboration features. Persistent file management is built-in, allowing users to maintain versioned script libraries and integrate custom code into system components.

SVG Single Line Diagrams

- Initially developed as a conceptual prototype, the SVG diagram component has matured into a tool for visualizing electrical system layouts. It supports dynamic rendering of circuit diagrams, real-time overlays of status information, and customizable layouts. Users can save and toggle between multiple display modes to assist in diagnostics and training.

Market Dashboard

- The Market Dashboard introduces a centralized interface for DER Aggregators and market-facing users. It consolidates performance data, bidding opportunities, and portfolio insights into an actionable format. This dashboard bridges operational insights with energy market participation, laying the foundation for automated trading and dispatch tools.

This current implementation demonstrates that our design approach prioritizing modularity, real-time interactivity, and user-centered control is both feasible and adaptable. However, continued refinement and feature expansion are needed to reach production-grade deployment.

7 Ethics and Professional Responsibility

The development of GridAI involved significant ethical and professional responsibilities, given its critical role in power grid management and its potential impact on communities, energy providers, and the environment. Our team approached these responsibilities with deliberate care, considering how our design and implementation decisions would affect various stakeholders.

We understood that grid management software must prioritize reliability, security, and accessibility, while also supporting sustainable energy practices. Throughout the development process, we maintained a strong commitment to ethical principles, addressing concerns such as data privacy, system security, environmental impact, and social responsibility.

The following sections outlined how we incorporated these ethical and professional standards into specific aspects of the GridAI project.

7.1 Areas of Professional Responsibility/Codes of Ethics

Area Of Responsibility	**Definition**	**Relevant IEEE Code of Ethics Principle**	**Team Application**
Work Competence	Perform work of high quality, integrity, and professionalism. Continuously improve skills and knowledge.	To maintain and improve technical competence and to undertake only qualified tasks.	Our team members limit tasks to their expertise and seek expert guidance. We conduct code reviews and follow best practices to ensure high-quality deliverables.
Financial Responsibility	Deliver products of value at a reasonable cost and avoid conflicts of interest.	To avoid unlawful conduct and to reject bribery in all its forms(IEEE Code Ethics #3)	Ensured transparent budgeting of any cloud resources and avoided unnecessary expenditures.
Communication Honesty	Report information truthfully, clearly, and without deception.	To be honest and realistic in stating claims or estimates	Provided stakeholders with accurate project updates, acknowledged

			challenges, and communicated realistic timelines
Health, Safety, Well -Being	Prioritize the well-being of the public and ensure safety in all solutions.	To hold paramount the safety, health, and welfare of the public.	Consider safety features in the environment, test edge cases to prevent harmful recommendations, and comply with industry safety standards.
Property Ownership	Respect intellectual property and the contribution of others.	To give credit for intellectual property where appropriate	We cite all third-party libraries, follow open-source licenses, and acknowledge collaborators' contributions.
Sustainability	Protect the environment and use resources responsibly.	To strive to comply with sustainable development practices	Not applicable
Social Responsibility	Produce products that benefit society and communities fairly.	To improve the understanding of technology and its potential consequences	Designed our interface to be accessible to diverse users. Encouraging widespread use and positive societal impact.

Table 15 - Areas of Responsability Table

7.2 Four Principles

	Beneficence	Nonmaleficence	Respect for Autonomy	Justice
Public Health & Welfare	Improves grid reliability and stability in communities	Reduces operator risk by enhancing data accuracy	Enables customizable data visualization for operators	Ensures equal access to reliable energy and efficient solutions
Global/Cultural	Adapts to diverse user needs and device types	Ensures compatibility across devices and regions	Respect operator decisions and data privacy	Provide benefits to energy producers, operators, and consumers
Environmental	Encourage energy efficiency and renewable energy integration	Minimizes environmental impact with efficient tools	Offers eco-friendly operational options	Promotes sustainability and resource monitoring
Economic	Reduce of grid downtime, boosting productivity	Limits economic disruption for energy providers and consumers	Supports cost-effective solutions for all users	Offers market analytics tools and fair resource distribution

Table 16 - Four Principles Table

7.3 Virtues

- Integrity: We were honest, transparent, and consistent in our actions and communication.

  Team Action: We held weekly meetings to review progress honestly and share challenges. We did not conceal setbacks from stakeholders.

- Responsibility: We took ownership of tasks, met deadlines, and accepted accountability for outcomes.

  Team Action: Each member was assigned clear deliverables. If someone fell behind, they reported it, and we collectively found solutions.

- Collaboration: We worked cooperatively, respected diverse ideas, and supported each other's professional growth.

  Team Action: We held weekly programming sessions and offered constructive feedback to improve everyone's skills.

Virtue Demonstrated (as a team): Integrity

Importance: Integrity built trust and credibility in our work.

Demonstration: We established weekly check-ins to review work, share accomplishments, and address issues early, reinforcing integrity and responsibility.

Virtue Not Demonstrated(as a team): Time Management

Importance: Strong time management ensured consistent progress, reduced delays, and maintained team accountability.

Action: We proposed setting aside extra time at the beginning or end of weekly meetings to review tasks, track deadlines, and adjust workloads proactively but for different situations and the extent of the platform requirements it was difficult to allocate all the time needed to certain parts of the project.

8 Closing Material

8.1 Conclusion

The GridAI frontend enhancement project represents a significant advancement in modern power grid management interfaces. By addressing evolving user needs and integrating modern web technologies, our team transformed the original framework into a more robust, intuitive, and flexible platform. These enhancements introduced powerful tools that enable grid operators to monitor and respond to real-time conditions with greater accuracy and efficiency. As our development phase concludes, we leave a solid foundation for future teams to build upon, ready for continued refinement, advanced feature integration, and broader deployment in real-world grid environments.

8.2 Value Provided

Our contributions significantly enhanced GridAI's functionality, usability, and extensibility. Key deliverables include:

- A dynamic Widget Dashboard with live data support, drag-and-drop layout, and a built-in Widget Editor
- Real-time communication integration using WebSockets and backend event streams via Kafka
- Support for custom widgets, role-specific interfaces, and historical data playback
- A refined frontend architecture with modular components, responsive design, and persistent widget configuration via Firebase
- A working Market Dashboard, SVG single-line diagram visualizations, and collaborative code editor

These features not only modernize the platform but also position it for real-world scalability and continued research.

8.3 Next Steps for Future Developers

To ensure GridAI's long-term viability and readiness for deployment in production environments, we recommend the following strategies for the next development phase:

1. **Implement Multi-Node Comparison Widgets**
   Introduce additional widget templates or functionality within the Widget Editor that allow users to compare telemetry from two node keys simultaneously. This functionality is already implemented in the backend and only requires frontend integration. Adding support for side-by-side or overlay comparison will significantly improve monitoring and diagnostic utility.

2. **Expand Historical Data Integration Using InfluxDB**
   Leverage the existing InfluxDB backend to implement robust time-series data storage and querying capabilities. Enabling widgets to request historical data directly from specific widgets components will improve diagnostics, forecasting, and post-event analysis.

3. **Enhance Role-Based Access Control**
   Refine the existing role-based permission system to enforce granular access control for operators, developers, and administrators. Secure sensitive features such as widget editing or code modification based on user roles, minimizing risk and aligning with operational security policies.

4. **Enhance Testing and Error Monitoring**
   Increase test coverage for both frontend components and backend data pipelines. Increase the amount of automated unit and integration tests, especially around WebSocket and Kafka messaging. Add frontend runtime error reporting and logging to assist in debugging and system monitoring.

5. **Strengthen Documentation and Onboarding**
   Develop comprehensive technical documentation, including architecture overviews, API references, setup guides, and best practices. Create onboarding materials tailored for future student teams or project contributors to reduce ramp-up time and ensure smooth project handovers.

9 Acknowledgment

This research is partially supported by the U.S. NSF Grant # CNS-2105269, U.S. DOE CESER Grant DE-CR000016, and the Iowa Energy Center Grant #21-IEC-009.

10 References

[1] IEEE. "IEEE 90003:2018: Software Engineering Guidelines for Application." IEEE Standards Association, 2018.

[2] Benson, K., et al. "Modern Web Application Architecture: A Comprehensive Analysis." International Journal of Software Engineering, vol. 15, no. 2, 2023, pp. 45-67.

[3] Smith, J., and Johnson, R. "Real-time Data Visualization Techniques for Power Grid Management." IEEE Transactions on Power Systems, vol. 38, no. 4, 2023, pp. 3456-3470.

[4] Firebase. "Firebase Documentation." Google Firebase, 2024, https://firebase.google.com/docs.

[5] React. "React Documentation." Meta Open Source, 2024, https://react.dev/.

[6] InfluxDB. "InfluxDB Documentation." InfluxData, 2024, https://docs.influxdata.com/.

[7] OpenDSS. "OpenDSS Manual." Electric Power Research Institute, 2023.

[8] Williams, P., and Davis, M. "Best Practices in Power Grid Visualization." Power Systems Engineering Journal, vol. 42, no. 3, 2023, pp. 89-102.

[9] Anderson, R., and Lee, S. "User Interface Design for Critical Infrastructure Systems." Journal of Critical Infrastructure Protection, vol. 18, 2023, pp. 234-248.

[10] Taylor, A., et al. "Modern Frontend Architecture Patterns." Software Architecture Journal, vol. 29, no. 1, 2024, pp. 12-28.

11 Appendices

APPENDIX 1 – OPERATION MANUAL

GridAI

IMPORTANT

- Please run this command every couple of weeks, Docker does not have automatic garbage collection, this can result in your disk storage depleting if not cleaned of old containers and images.
    - `docker system prune` (normal clean, will not remove latest containers and built images)

    - `docker system prune -a` (for full, deep clean including latest images and containers)

Project Setup

- Execute presetup script passing the full path to your project root and follow prompts:
    - `./scripts/presetup.sh [proj path]`
- Make sure you have Docker and it's related packages installed:
    - `sudo ./scripts/setup.sh`
- Restart your system, this is necessary to add your user to the Docker group and allow you to use it without root:
    - `sudo restart now`
- Test that Docker is installed correctly and can be run without root:
    - `docker -v`

Widget Lifecycle

The full lifecycle of a widget in GridAI, from creation to real-time streaming:

1. Fetching Widgets

- `AllWidgets.tsx` calls `WidgetState.getWidgets()` to retrieve all user widgets.
- Data is fetched from Firestore via the backend `/widgets` API.
- Widgets are displayed in a sortable, editable table.

2. Creating a Widget

- `AddWidgetForm.tsx` provides a form where users can select from predefined templates (`WidgetTemplates.ts`).
- On submission, `WidgetState.addWidget()` creates the new widget in Firestore.
- The widget is immediately available for editing or dashboard assignment.

3. Editing a Widget

- Clicking "Edit" redirects to `[widgetId]/page.tsx` (Widget Editor).
- Users can modify:
    - Template code (`templateHtml`)
    - Controller script (`controllerScript`)
    - Widget title and description
    - Node Key, Measurement type, and Data Field (for telemetry)
- Live preview updates after saving changes.
- Clicking "Save Changes" calls `WidgetState.updateWidget()` to save updates back to Firestore.

4. Rendering a Widget

- Widgets are compiled dynamically at runtime using Babel (`@babel/standalone`).
- `templateHtml` (JSX) is transformed into a React component.
- `controllerScript` is injected as a global function (`window.controllerScript`) to manage widget logic.
- Errors during rendering are caught and displayed using an `ErrorBoundary`.

5. Subscribing to Live Data

- Clicking "Subscribe" inside the Widget Editor with a valid Node Key:
  - Connects to the backend `/live-data` WebSocket server.
  - Listens for live Kafka telemetry for the selected `nodeKey`.
  - Updates the widget preview with real-time streaming data (e.g., voltage, real/reactive power).

6. Switching Node Keys

- Clicking "Switch Key":
  - Disconnects the previous WebSocket connection and unsubscribes from the old telemetry stream.
  - Updates the widget's internal nodeKey and recompiles it.
  - Reconnects to Kafka telemetry for the new nodeKey.
  - Live data continues updating without page reload.

7. Unsubscribing

- Clicking "Unsubscribe":
  - Disconnects the WebSocket connection.
  - Stops receiving live Kafka telemetry updates.
  - Clears live telemetry history for that widget.

8. Deleting a Widget

- In `AllWidgets.tsx`, clicking "Delete" calls `WidgetState.deleteWidget()`.
- Deletes the widget from Firestore.
- UI automatically refreshes to remove the deleted widget from the list.

APPENDIX 2 – ALTERNATIVE/INITIAL VERSION OF DESIGN

In the early stages of development, widget data for GridAI was originally stored using MongoDB, integrated with a custom backend API for create, read, update, and delete (CRUD) operations.

However, as the system evolved, we transitioned to Firebase to simplify real-time updates, enable native frontend integration, and reduce the need for managing backend infrastructure.

The final implementation uses Firebase Firestore as the primary database for widget storage, offering built-in support for document syncing, authentication, and timestamping. Widget data includes fields such as title, type, descriptor, and editable properties like templateHtml, templateCss, and controllerScript. Each widget is uniquely identified by a widgetId and scoped by organization and user identifiers.

The decision to shift from MongoDB to Firebase was instrumental in supporting real-time updates and improving the development workflow within a React-based frontend environment.

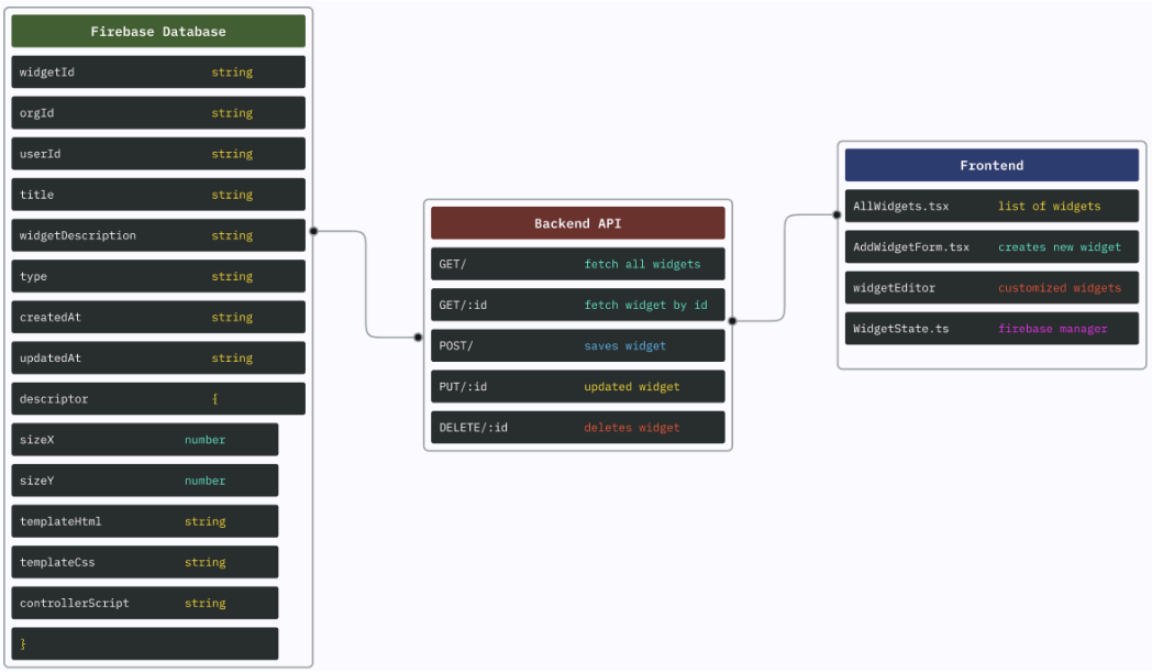

Figure 7 - Previous Widgets Architecture Diagram

APPENDIX 3 – OTHER CONSIDERATIONS

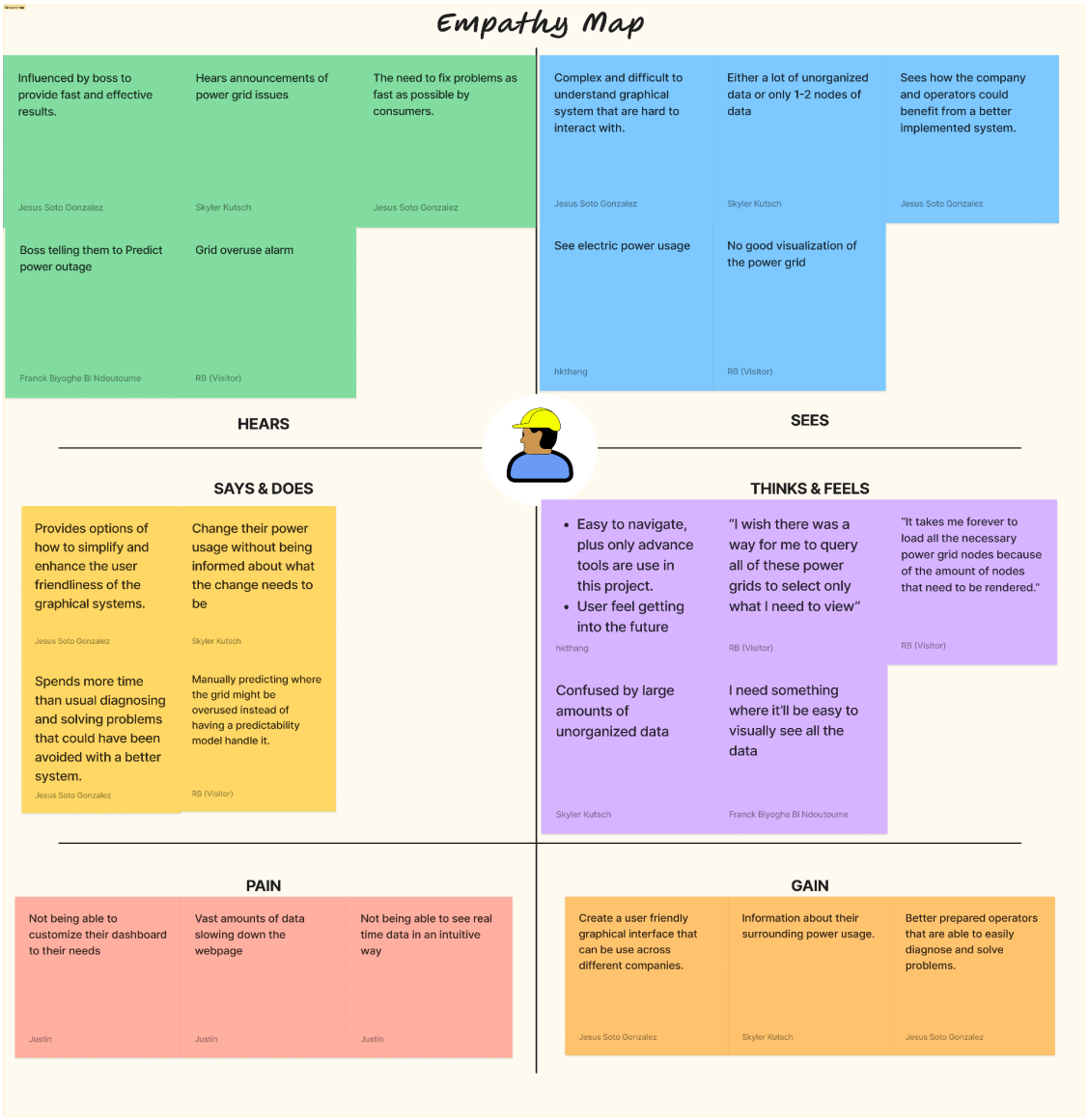

Figure 8 - Empathy Map

Widget Descriptor Schema

Each widget in the system is stored with the following Firestore schema:

```
interface Widget {
  widgetId: string;          // Unique ID for the widget
  orgId: string;             // Organization ID
  userId: string;            // User ID (owner of the widget)
```

```
  title: string;              // Widget title (e.g., "Live Voltage Monitor")
  widgetDescription: string;  // Short description for users
  type: string;               // Widget type (e.g., "Line Chart", "Gauge")
  createdAt: string;          // Timestamp when widget was created
  updatedAt: string;          // Timestamp when widget was last updated
  descriptor: {
    sizeX: number;            // Grid width in dashboard layout
    sizeY: number;            // Grid height in dashboard layout
    templateHtml: string;     // JSX/HTML template for the widget body
    templateCss: string;      // CSS (future: dynamic styling)
    controllerScript: string; // Script to handle telemetry subscription/logic
    nodeKey?: string;         // Node Key (Kafka) to subscribe for live data
    measurement?: string;     // Measurement type (e.g., "power", "voltage")
    dataField?: string;       // Specific field within the measurement ("p", "q", "kv")
  };
}
```

Widget Interaction with Dashboards

Widgets are created and stored independently but can be dynamically linked to user dashboards.

- Each dashboard maintains a list of associated widget IDs.
- When a dashboard loads, it fetches its associated widgets and renders them in a grid layout.
- Widgets live inside a grid system where their size (`sizeX`, `sizeY`) and position are customizable.
- Live widgets automatically subscribe to telemetry data after being placed inside dashboards.
- Currently users can change the Node Key in the Widget Editor and save to see new node in dashboard.

Key Functions:

- `DashboardState.addWidgetToDashboard(dashboardId, widgetId)` - Associates a widget with a dashboard.

- `DashboardState.removeWidgetFromDashboard(dashboardId, widgetId)` - Disassociates a widget from a dashboard.
- `DashboardState.updateWidgetLayout(dashboardId, layoutConfig)` - Updates widget size and position inside the dashboard grid.

APPENDIX 4 – CODE

GitLab Repository: https://git.ece.iastate.edu/sd/sdmay25-43

Widget Component File Structure

Frontend (`ui/frontend/src/app/derms/widgetLibrary/`)

widgetLibrary/

```
│
├── [widgetId]/              # Dynamic route for editing a specific widget
│   ├── page.tsx             # Full Widget Editor (template, controller, settings, preview)
│   └── ErrorBoundary.tsx    # Error handler for widget rendering
│
├── allWidgets/              # Manages all widgets list and creation
│   ├── AllWidgets.tsx       # Lists all widgets (with edit/delete)
│   ├── AddWidgetForm.tsx    # Form to create a new widget
│   └── WidgetTemplates.ts   # Predefined widget templates
│
├── state/
│   └── WidgetState.ts       # Handles fetching, adding, updating, deleting widgets
│
├── testmosaic/
│   └── widgetEditor.module.css  # Styles for Widget Editor quadrants
│
├── page.tsx                 # Main widget library landing page (tabbed layout for Widgets/Bundles)
├── README.md                # (This file)
```

Frontend (`ui/frontend/src/live-data/`)

live-data/

│

├── socket.ts # WebSocket client connector to live backend (/live-data)

├── telemetryManager.ts # Manages telemetry subscriptions (subscribe/unsubscribe to Kafka node keys)

Frontend (live-data/) → connects to → Backend (kafka/) → consumes from → Kafka Broker (topics)

Backend (`ui/backend/src/kafka/`)

kafka/

│

├── kafkaConsumer.ts # Kafka consumer that listens to topics and processes messages

├── liveDataServer.ts # Launches the backend WebSocket server for live widget telemetry

├── startLiveConsumer.ts # Starts the Kafka message consumer pipeline

├── telemetryFieldEmitter.ts # Extracts telemetry fields (e.g., voltage, real power, reactive power) from Kafka messages

├── liveDataSocket.ts # WebSocket server handling live widget connections at `/live-data`

│

├── routes/

│ └── proxy.ts # Proxy routes for interacting with Kafka services

│ ├── GET /project/id # Fetch project metadata for widgets

│ └── GET /partition # Get Kafka partition associated with a given node key

APPENDIX 5 – TEAM CONTRACT Team Contract

Team Members:

- Skyler Kutsch
- Frank Biyoghe Bi Ndoutoume

- Rangsimun Bargmann
- Jesus Soto
- Justin Soberano
- Hang Kim Thang

Team Procedures

- Day, time, and location (face-to-face or virtual) for regular team meetings:
    - In person or virtual if needed, Tuesdays 3-4 pm and Wednesdays 4:15-5:15 with the client.
- Preferred method of communication updates, reminders, issues, and scheduling (e.g., e-mail, phone, app, face-to-face):
    - Discord
- Decision-making policy (e.g., consensus, majority vote):
    - Majority vote
- Procedures for record keeping (i.e., who will keep meeting minutes, how will minutes be shared/archived):
    - Skyler will take meetings notes and record the amount of time per meeting.

Participation Expectations

- Expected individual attendance, punctuality, and participation at all team meetings:
    - Every member is expected to attend all meetings on time, and if issues arise and it is not possible to attend a meeting, they should try to be present using online tools.
- Expected level of responsibility for fulfilling team assignments, timelines, and deadlines:
    - Every member of the team is expected to complete their assigned tasks on time.
- Expected level of communication with other team members:
    - Every team member should share opinions and suggestions over the team discord at least once a week, especially during meeting days.
- Expected level of commitment to team decisions and tasks:
    - Every team member is expected to contribute to a team decision and commit to fulfilling their responsibilities to the best of their ability.

Leadership

- Leadership roles for each team member (e.g., team organization, client interaction, individual component design, testing, etc.):
    - Roles are subject to change, but for now:
        - Team Organization: Jesus Soto
        - Client interaction: Everyone
        - Role Manager: Rangsimun Bargmann
        - Test Lead: Franck
        - Cyber Security Lead: Hang Kim
        - Component Design: Justin Soberano
        - Record Keeper: Skyler Kutsch
- Strategies for supporting and guiding the work of all team members:

    - Frequently checking in with each member of the team to discuss any issues or roadblocks they might be facing.
    - Each team member posts and resolves a git issue before each weekly meeting
    - Sharing any new and relevant knowledge concerning different frameworks that we might need to use.
- Strategies for recognizing the contributions of all team members:
    - Discuss contributions to the project every week if time allows for meetings.
    - Recognizing and valuing various forms of contribution, such as coding,
    - generating ideas, assuming responsibilities, creating diagrams, and more.

Collaboration and Inclusion

- Describe the skills, expertise, and unique perspectives each team member brings to the team.
    - **Franck**: 1 year Full-stack as a part-time student at John Deere.
    - **Hang**: Inspect a product or software in very detail, make sure there is no error or a bad one on the project.
    - **Justin**: Software engineering major with experience in working in Full-stack projects ranging from personal to industry experience.
    - **Skyler**: Previous projects with React web design and frontend programming
    - **Rangsimun**: Software engineering major, minoring in cyber security engineering. Experience working as an engineering analyst in industry and developing a React web app.
    - **Jesus**: Software engineering major with experience in web development.
- Strategies for encouraging and supporting contributions and ideas from all team members:
    - Short quick updates if needed at the end of 4910 class if any tasks require the attention or input of another team member or to just update each other.
    - Develop an open environment that encourages ideas to be expressed and acknowledged.
    - Meet in person if any of our tasks require extensive collaboration
- Procedures for identifying and resolving collaboration or inclusion issues (e.g., how will a team member inform the team that the team environment is obstructing their opportunity or ability to contribute?)
    - Communicate with the team member individually or with the whole group
    - Talk to the professors or the group advisor if needed to resolve the issue

Goal-Setting, Planning, and Execution

- Team goals for this semester:
    - Have a well-organized and structured project design with specific team roles and responsibilities.
    - Start the development and enhancement of required components.
    - Meet all the requirements for the project
    - Become better frontend developers

    - Finish project design and ready to build it next semester.
- Strategies for planning and assigning individual and team work:
    - Distribute work assignments based on experience and personal interests.
    - Each team member posts and solves a git issue before the weekly meeting
- Strategies for keeping on task:
    - Weekly team progress checks.
    - Attending weekly team meetings.
    - Posting git issues
    - Checking Discord messages and responding in a timely manner

Consequences for Not Adhering to Team Contract

- How will you handle infractions of any of the obligations of this team contract?
    - Problems will be dealt with on a case-to-case basis, with increasing consequences if patterns emerge.
- What will your team do if the infractions continue?
    - Contact the advisor

****

a) I participated in formulating the standards, roles, and procedures as stated in this contract.

b) I understand that I am obligated to abide by these terms and conditions.

c) I understand that if I do not abide by these terms and conditions, I will suffer the

consequences as stated in this contract.

1) Franck Biyoghe Bi Ndoutoume	DATE
2) Justin Soberano	DATE 12/7/24
3) Rangsimun Bargmann	DATE 12/7/24
4) Jesus Soto	DATE 12/7/24
5) Skyler Kutsch	DATE 12/7/24
6) Hang Kim Thang	DATE 12/7/24