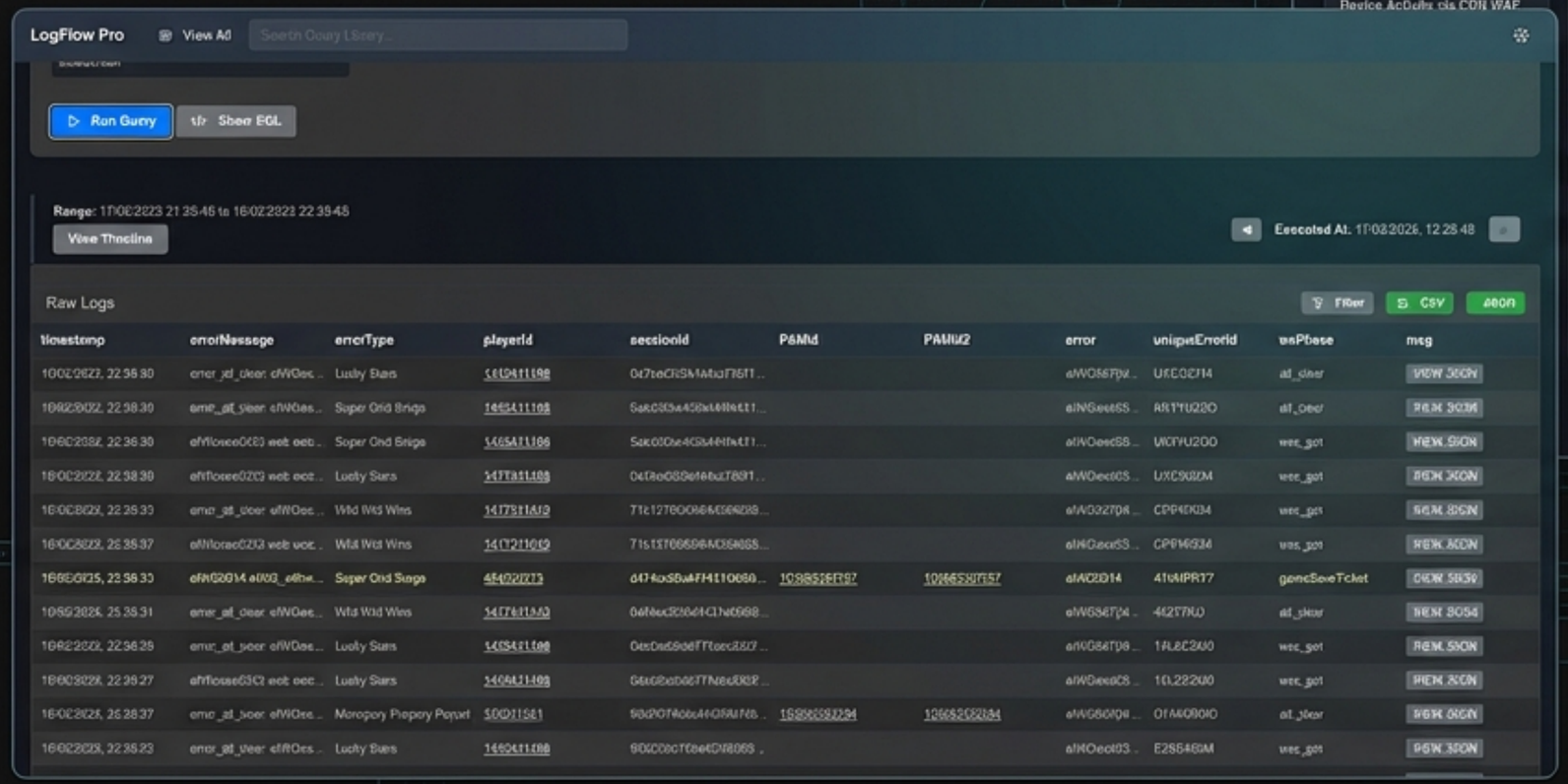
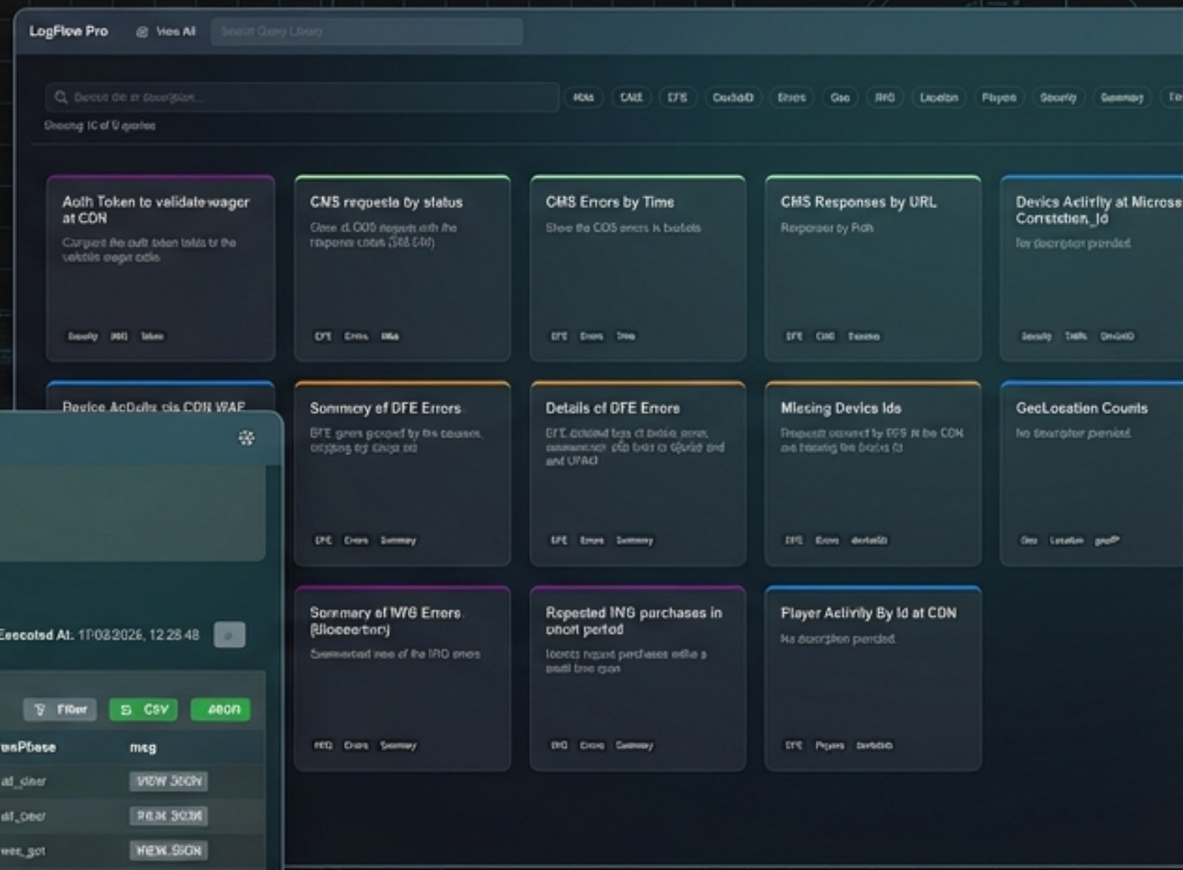
LogFlow Pro v1.0

The bridge between raw AWS CloudWatch data and actionable engineering insight.





Advanced diagnostic platform.
Stateless Backend. Stateful Frontend.

From Ad-Hoc Chaos to Structured Intelligence

The Problem: Raw CloudWatch

Unsaved, repetitive queries
`SELECT * FROM "logs" WHERE message LIKE '%error%'`

Manual entry fatigue
Constantly re-typing `filter @message like /Request failed/`

Lack of historical context
No view of previous investigations or `timestamp > -1h` awareness.

Disjointed workflows
Switching between consoles, terminals, and docs.
`aws logs filter-log-events --log-group-name /aws/lambda/my-function`

The Solution: LogFlow Pro

Structured Query Management
Centralized repository for `saved_queries.json`.
`GET /api/queries/Q-1234`

Reusable assets
Parameterized queries like `filter @message $` and shared snippets.

Persistent History
Audit trail of all executed queries and results with
`{"timestamp": "2023-10-27T10:30:00Z", "user": "admin"}`

Integrated Visualization
Embedded charts and graphs directly from
`logflow-pro/visualization-component`

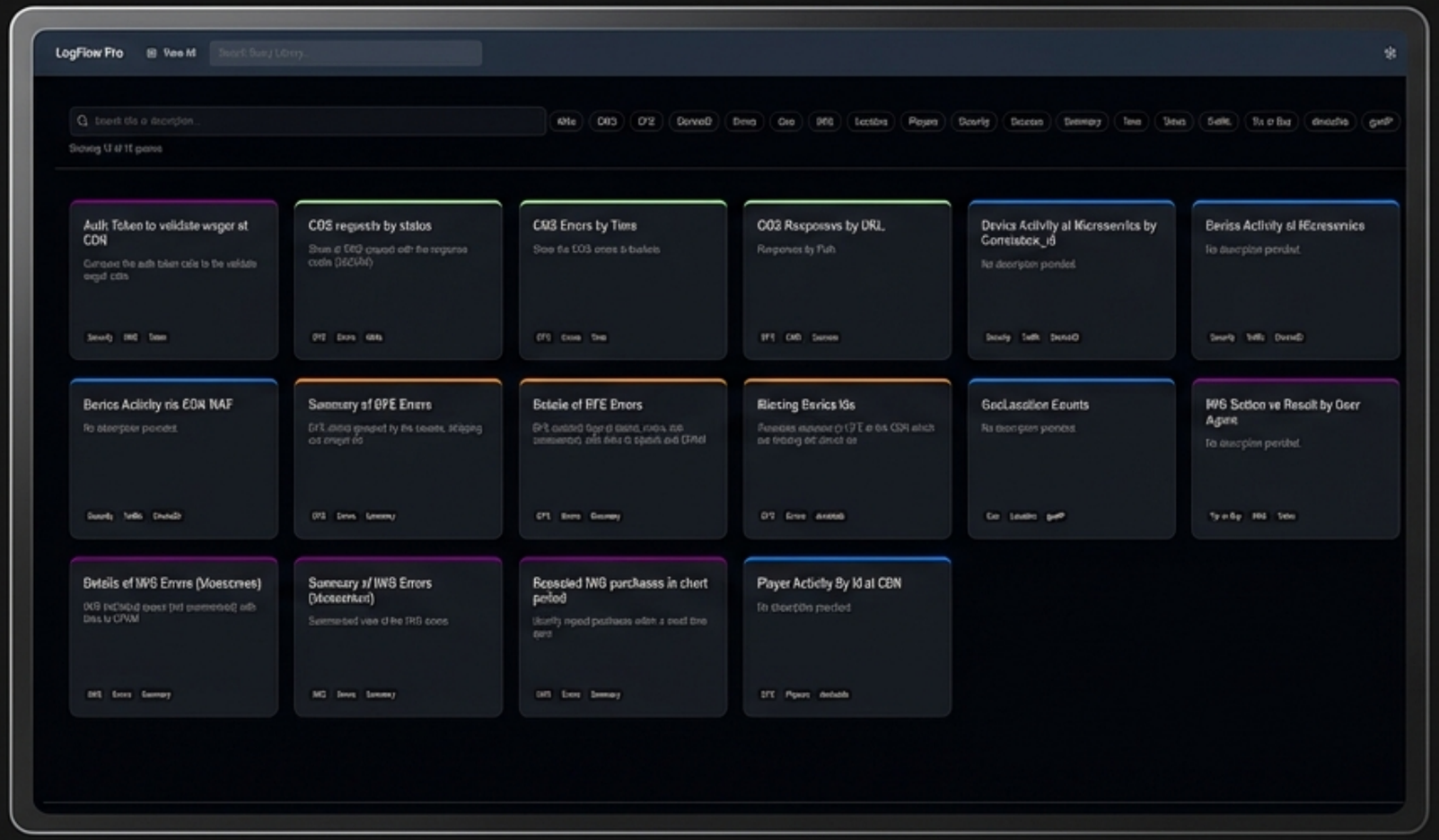
System Architecture



Philosophy: Offloading state and visualization to the client for responsive, rich user experiences. `Client-Side Processing > Server-Side Aggregation`

A Centralized Query Library

Technical Blueprint meets High-Fidelity Software UI



Tag-Based Filtering

Organize by functional area (e.g., DFE, Errors, Security).

Visual Cues

Color-coded borders defined in metadata for rapid identification.

Searchability

Instant browsing by title, description, or tags.

Takeaway

Move beyond ad-hoc searching. Build a shared, version-controlled library of engineering knowledge.

Configuration as Code: The DSL

query.txt in JetBrains Mono

```
Ref: SAMPLE-01
Title: Example Error Query
Region: eu-west-2

Parameters: level=text:ERROR:Find specific log level
Query: fields @timestamp, message | filter level = '{{level}}'

Views: [Log Level Distribution] GroupBy: level
```

Metadata (Targeting Infrastructure)

Dynamic Inputs

Raw AWS CloudWatch SQL

Client-side Aggregation Rules

Technical Blueprint meets High-Fidelity Software UI

Dynamic Parameter Engine



Syntax & Validation

Uses {{key}} placeholders.

Mandatory fields marked with * block execution if empty.

Smart Omission

Optional fields (text-opt) trigger logic checks. If left blank, the backend removes the entire line of SQL containing that placeholder.

Benefit

Prevents SQL syntax errors without complex conditional logic.

Mastering Time: The SmartRangeParser

Smart Range Input

last 2 hours

Smart Range used: "last 2 hours"



Natural Language Features

- Rolling Periods: `"last 2 hours"` -> `Dynamic lookback.`
- Calendar Snapping: `"the last hour"` -> `14:00 to 14:59.`
- Complex Ranges: `"yesterday 5pm to now".`

The Architecture of Execution

Client (Browser)

Emits `StartQueryCommand`

Server (Node.js)

Polling Loop (2s interval).
`GetQueryResultsCommand`

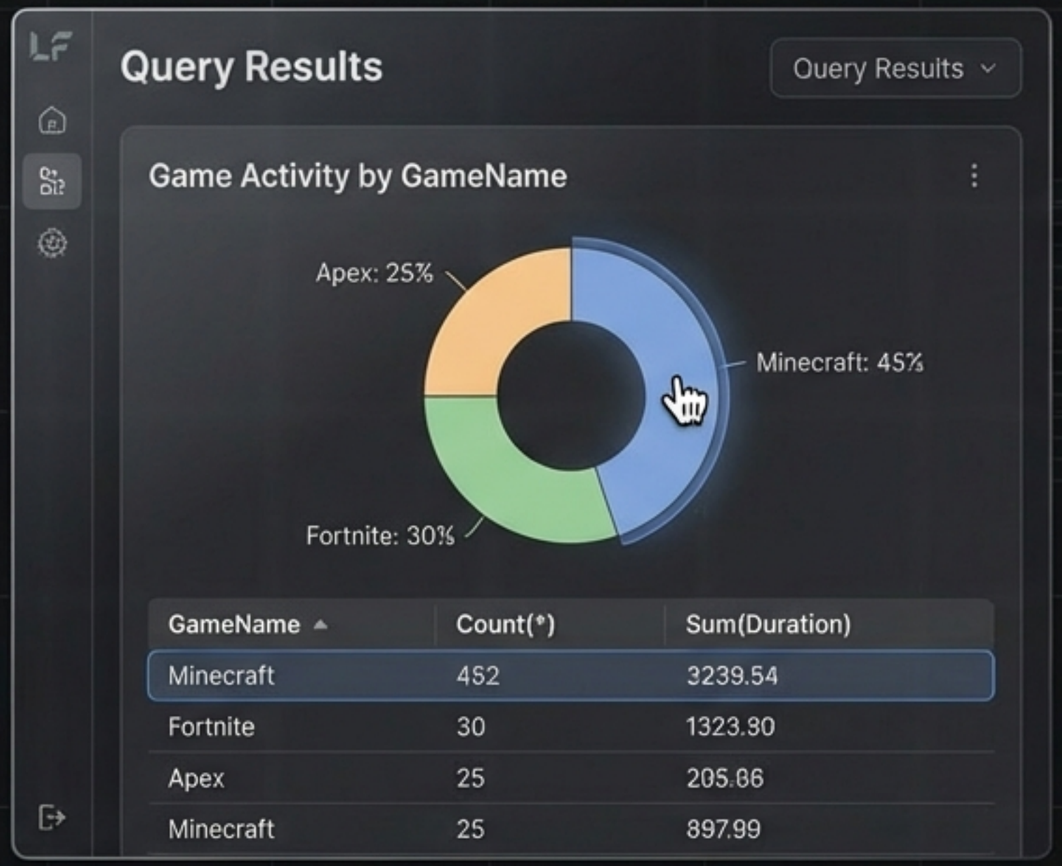
WebSocket (Port 8080)

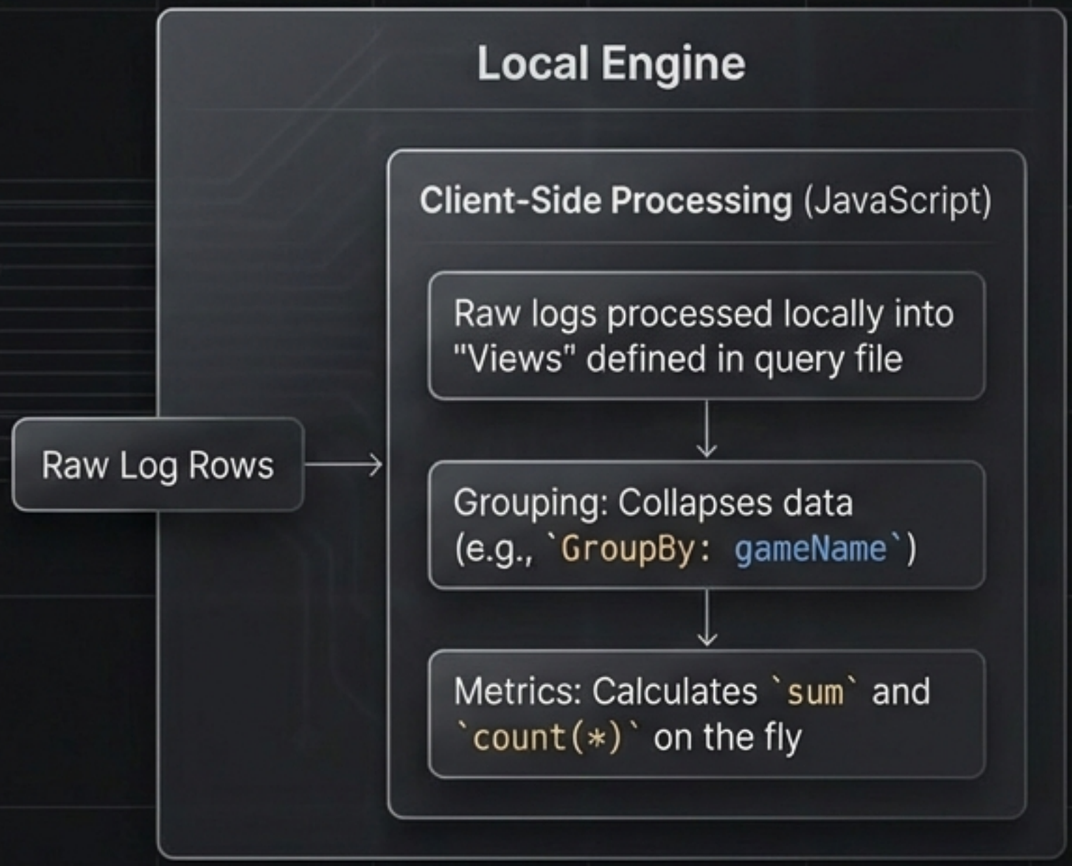
Pushes Real-time Status:
`Running` -> `Scheduled` -> `Complete`

Self-Healing SSO Credential Guard

If AWS SSO authentication fails, the backend executes `process.exit(1)`. This hard stop signals the engineer to run `aws sso login` immediately.

Client-Side Aggregation & Visualization





Interactivity: Clicking a chart segment filters the data table instantly without re-querying the server.

Interactive Data & Deep Linking

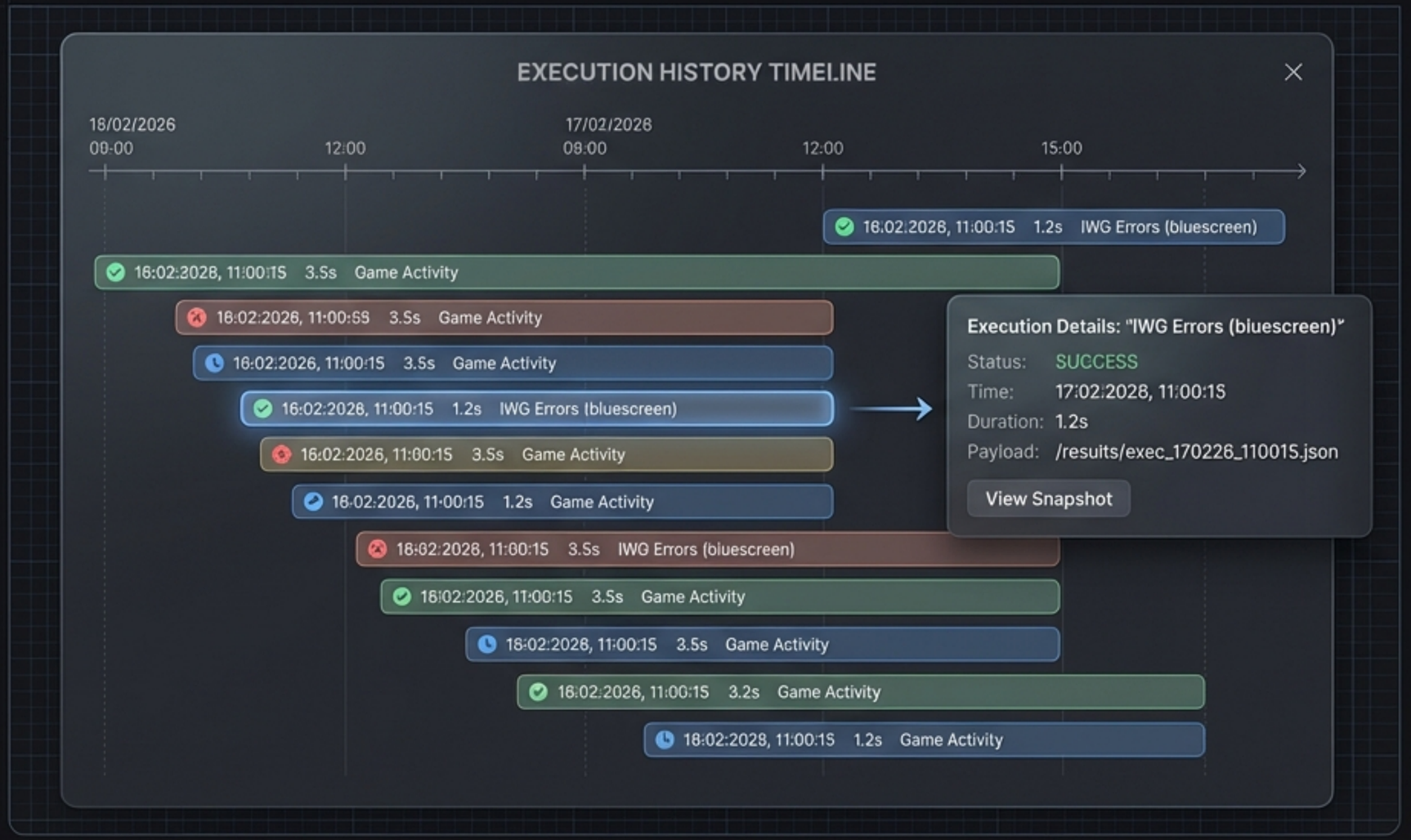


HREF Formatter:
Deep-links to external tools (`Splunk`, `Back Office`) using row context.

JSON Formatter:
Opens payload in interactive `JsonTree.js` viewer.

Automatic formatting
for readability.

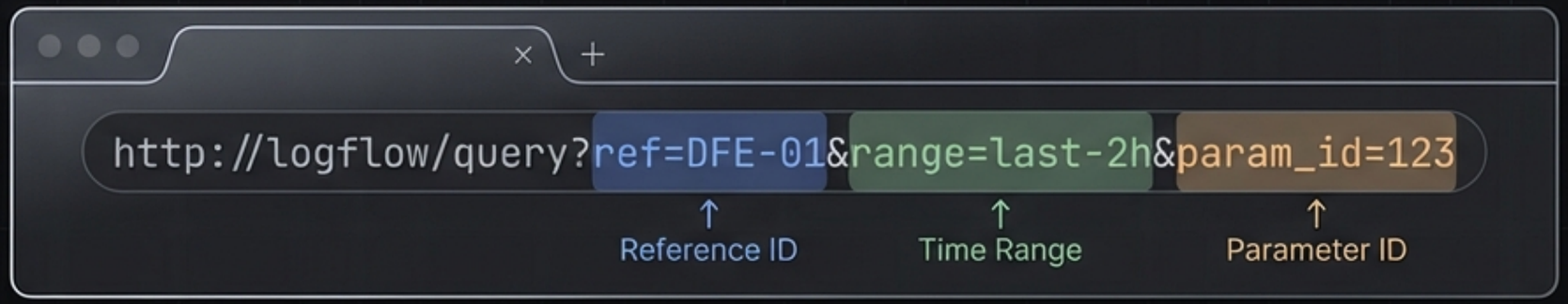
Historical Intelligence: The Timeline



FEATURE HIGHLIGHTS

- **The God-View**: Visual SVG modal showing execution history as bars on a time-axis.
- **Persistence**: Executions are stored as JSON snapshots in the `/results` directory (e.g., `/results/exec_170226_110015.json`).
- **Payload**: Includes metadata, time context, and full data for every run.
- **Philosophy**: Why re-run a query when the snapshot already exists? Access historical data instantly.

State Persistence & Sharing



State Capture

`pushState` captures every parameter change and time range.

Collaboration

Share the link to load the exact 'frozen' analysis state for a colleague.



JSON Snapshot

Automatic audit trail: Every shared state generates a JSON snapshot for historical reference.

Technical Stack & Dependencies

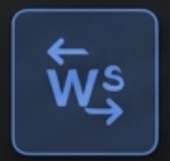
Core Backend



Node.js (v18.x): Runtime environment.



Express: Web server framework.



WS: WebSocket library for real-time heartbeats.

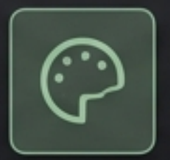


AWS SDK: @aws-sdk/client-cloudwatch-logs.

Frontend & UX



Chart.js: Data visualization.



Phosphor Icons: Modern, scalable iconography.



JsonTree.js: Payload inspection.

Guide to Query Creation

query.txt

1. Headers
Ref, Title, Tags, Region, LogGroups.

2. Parameters
Define inputs (key=type:default:hint).

3. Query
The SQL logic block.

4. Formatters
Column rendering rules (col:type:format).

5. Views
Aggregation definitions ([View Title], Charts).

Files located in ./queries

Installation & Deployment

1. Setup

Clone repository. `npm install`.

2. Authenticate

Ensure environment is authenticated via `aws sso login` (**Critical**).

3. Execute

Run `node app.js`.

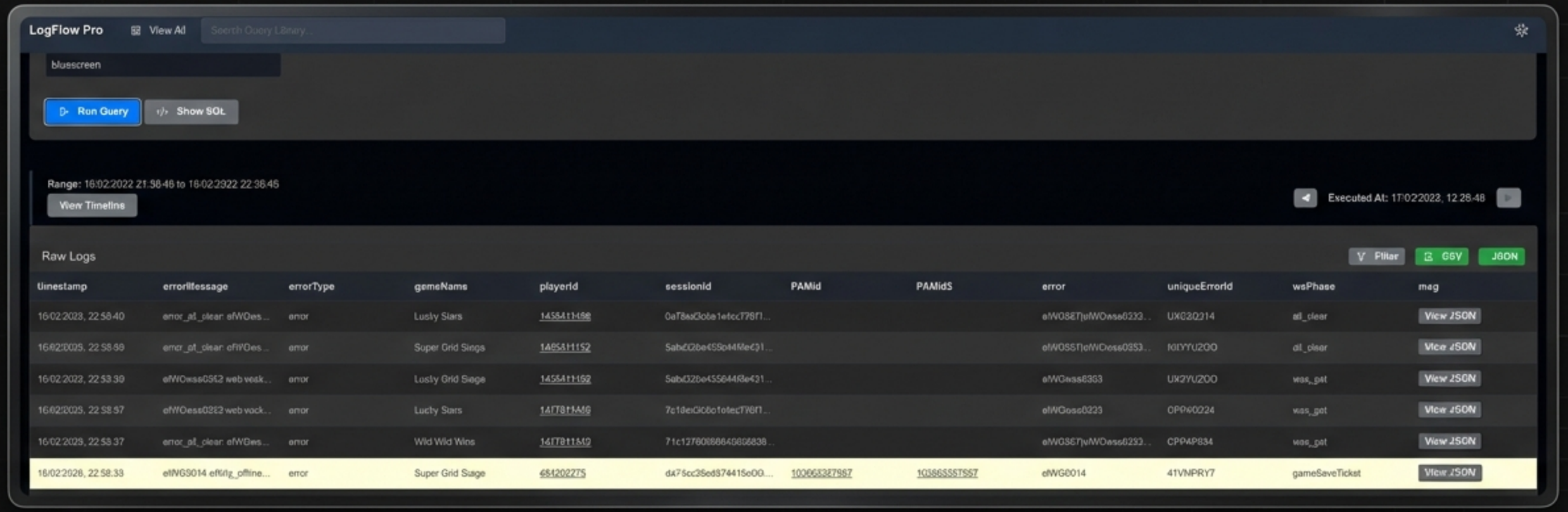
Configuration Details

UI Port: Defaults to 3001.

WebSocket: Hardcoded to 8080.

Directories: Queries load from `./queries`; results save to `./results`.

The Future of Log Analysis



STRUCTURED. INTELLIGENT. PERSISTENT.

LogFlow Pro: Turning logs into logic.