Intelligent Alert System – Case Study Submission

1. Introduction

This project is an **"Intelligent Alert System"**. Its primary function is to monitor incoming events and transform them into actionable alerts. Each alert is assigned a severity: CRITICAL, WARNING, or INFO. The web dashboard allows users to view these alerts in real-time, filter them, read details, and resolve them. Additionally, a lightweight rule engine helps determine the severity when an alert is first created.

Why this matters

In many real systems such as fleet tracking, health monitoring, or application logs you get a lot of raw events. You cannot show all of them to users. You group them as alerts that carry important meaning, like: "Driver exceeded speed", or "Sensor offline". This project shows the basic pattern for that.

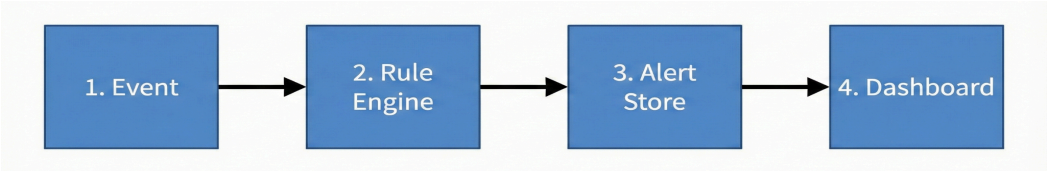
2. Problem Statement

We need a small demo system where events from different sources become alerts. We should be able to:

1. **Create alerts** from events automatically.
2. **Assign a severity** so a user knows how urgent it is.
3. **Change alert status** over time (OPEN, AUTO-CLOSED, RESOLVED, maybe ESCALATED later).
4. **See alert history** (how it changed and when).
5. **Filter by severity** so the screen is not too noisy.
6. **Show some simple stats** (counts, top drivers, trends) to feel like a dashboard.

3. High-Level Flow (Step by Step)

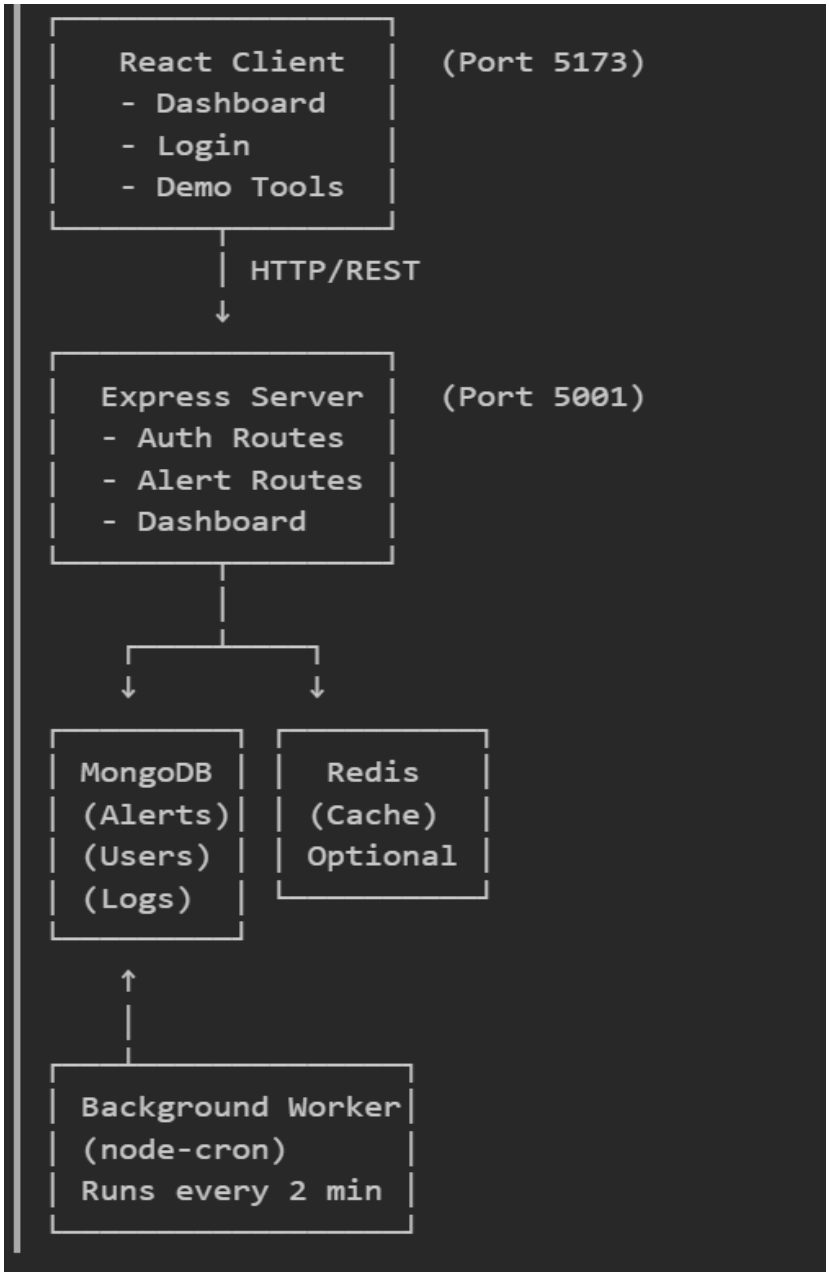
1. An event comes in (or is simulated by a demo tool).
2. The backend checks the event against rules.
3. A severity is selected (for example CRITICAL if it matches a special rule).
4. The alert gets saved into MongoDB with a history record.
5. The frontend asks the backend for a list of recent alerts every few seconds.
6. The user clicks an alert to see more details and full history.
7. The user can resolve the alert manually (adds a reason and changes status).
8. Background logic may auto-close some alerts if rules say so.



High-Level Flow

4. Architecture

- **Event Source:** A simulator or any external system.
- **Express API:** Receives events and exposes endpoints.
- **Rule Engine:** Decides severity and maybe auto-close later.
- **MongoDB:** Stores alert documents.
- **React Frontend:** Dashboard the user sees.



Visual Representation of System Architecture

5. Tech Stack Summary

- **Backend:** Node.js + Express + TypeScript
- **Database:** MongoDB (Mongoose models)
- **Frontend:** React (with Vite build tool)
- **Auth:** Simple token kept in sessionStorage
- **Security helpers:** Helmet, CORS
- **Other tools:** dotenv for environment variables, Passport init scaffolding

6. Data Model

Each alert stores:

- alertId: Unique id so we can find it again.
- sourceType: What kind of source (like driver, sensor, system).
- severity: How serious it is (CRITICAL, WARNING, INFO).
- timestamp: When it happened.
- status: Current state (OPEN, AUTO-CLOSED, RESOLVED, maybe ESCALATED later).
- metadata: Extra info (example: driverId, speed, etc.).
- history: List of state changes with time and optional reason.

Code Snippet (Model):

```
// backend/src/models/Alert.ts
const AlertSchema = new Schema<IAlert>({
  alertId: { type: String, required: true, index: true, unique: true },
  sourceType: { type: String, required: true, index: true },
  severity: { type: String, required: true },
  timestamp: { type: Date, required: true, index: true },
  status: { type: String, required: true, default: "OPEN" },
  metadata: { type: Schema.Types.Mixed, default: {} },
  history: [{ state: String, ts: Date, reason: String }],
  expiryTimestamp: { type: Date },
  lastTransitionAt: { type: Date },
  lastTransitionReason: { type: String },
});
```

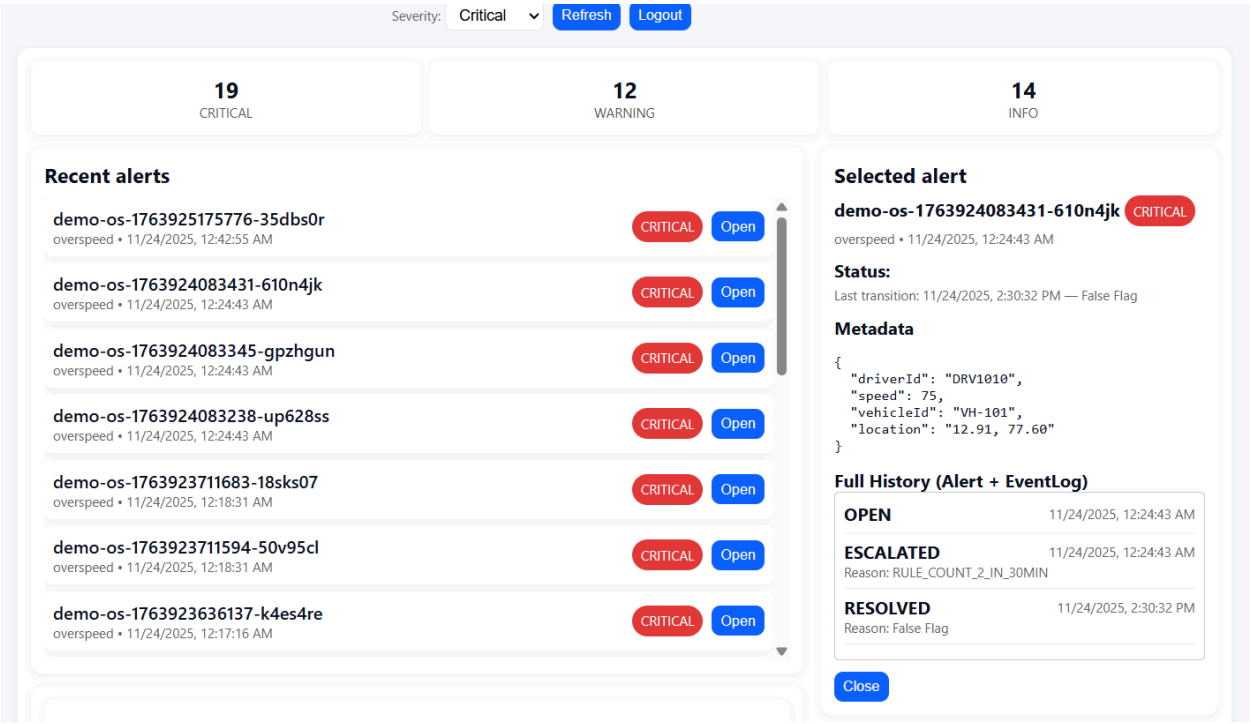
7. Features Implemented

- Create alerts and set severity using rules.
- Filter alerts by severity.
- View full history (states + events).
- Manual resolve with optional reason.

- Auto-refresh without losing filter.
- Admin-only rule reload.
- Basic stats: counts, top drivers, trends.
- Auto-closed alerts list.

8. Sample User Journey

1. User opens site and logs in.
2. Sees list of alerts.
3. Changes severity filter to CRITICAL to focus.
4. Clicks one alert, reads metadata and timeline.
5. Types a reason and resolves it.
6. After refresh, alert shows RESOLVED status.



A critical alert resolved manually

9. Backend Startup

When the server starts it:

1. Loads environment variables.
2. Connects to MongoDB.
3. Loads rules into memory.
4. Starts a worker for background tasks.
5. Listens on a port.

Code Snippet:

```
// backend/src/index.ts (Sample)
app.use('/api/alerts', alertsRouter);
app.use('/api/auth', authRouter);
app.use('/api/events', eventsRouter);
app.use('/api/dashboard', dashboardRouter);

await connectDB();
await loadRules();
startWorker();
```

```
[nodemon] starting `node --import tsx src/index.ts`
[dotenv@17.2.3] injecting env (10) from .env -- tip: ⚙️ suppress all logs with { quiet: true }
[dotenv@17.2.3] injecting env (0) from .env -- tip: 🔐 encrypt with Dotenvx: https://dotenvx.com
Connected to MongoDB
Loaded 21 rules
[worker] scheduled with cron: */2 * * * *
Server started on port 5001
Loaded 21 rules
[worker] running at 2025-11-24T08:10:00.029Z
[worker] running at 2025-11-24T08:12:00.029Z
[worker] running at 2025-11-24T08:14:00.025Z
[worker] running at 2025-11-24T08:16:00.032Z
[worker] running at 2025-11-24T08:18:00.024Z
```

10. Creating an Alert (Logic)

When an event comes in we evaluate it against rules. A matching rule can set severity or add a reason. Then we save the alert and log a created event.

Code Snippet:

```
// backend/src/services/alertService.ts (excerpt)
export async function createAlert(payload: any): Promise<IAlert> {
  const now = new Date();
  const evaluation = await evaluateEventAgainstRules(payload);

  const toSave = {
    alertId: payload.alertId,
    sourceType: payload.sourceType,
    severity: evaluation.severity,
    timestamp: payload.timestamp ? new Date(payload.timestamp) : now,
    metadata: payload.metadata || {},
    history: [{ state: 'OPEN', ts: now }],
    lastTransitionAt: now,
    lastTransitionReason: evaluation.matchedRule
```

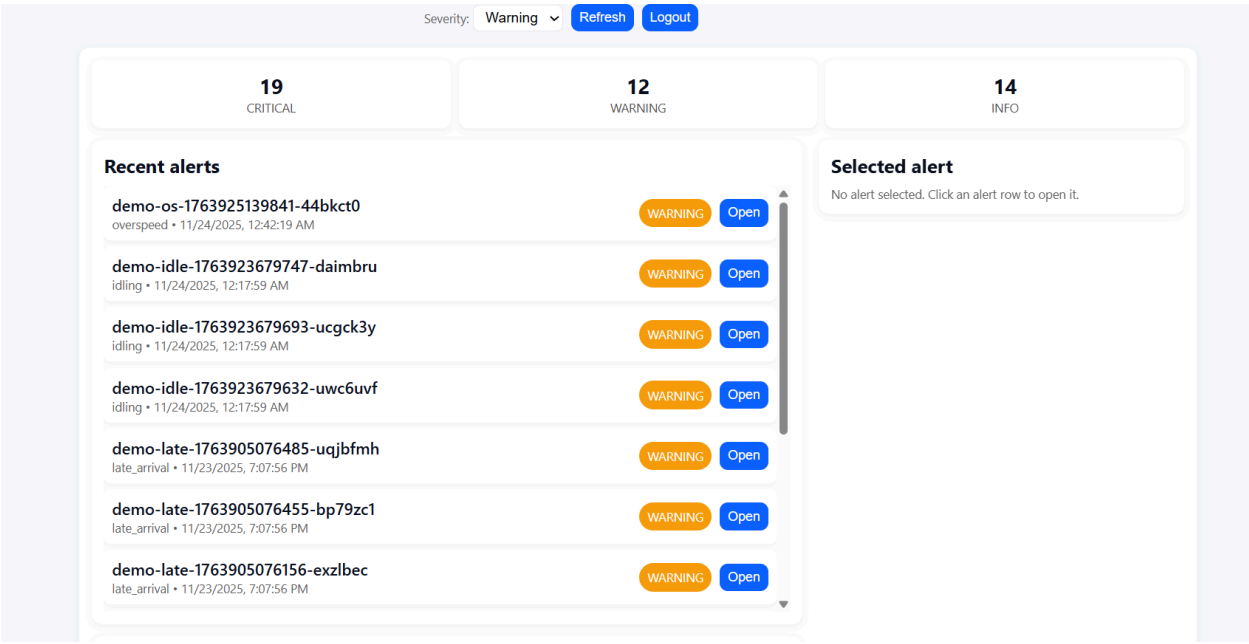
```
    ? `MATCHED_RULE:${evaluation.matchedRule.ruleId}`
    : 'CREATED',
  };

  const created = await AlertModel.create(toSave);
  return created;
}
```

11. Listing and Filtering Alerts

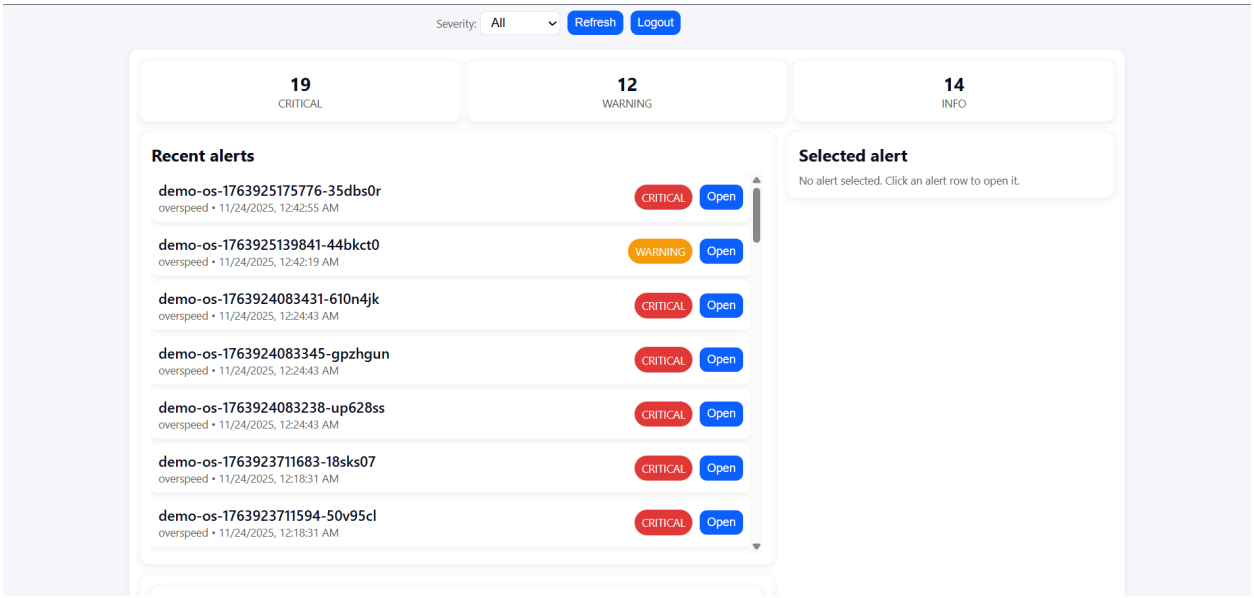
We can request alerts and pass filters. Severity is optional; if not passed we get all severities.

Code Snippet:

```
// backend/src/services/alertService.ts (excerpt)
export async function listAlerts(filters: any): Promise<IAlert[]> {
  const q: any = {};
  if (filters.status) q.status = filters.status;
  if (filters.sourceType) q.sourceType = filters.sourceType;
  if (filters.severity) q.severity = filters.severity;
  if (filters.driverId) q['metadata.driverId'] = filters.driverId;

  return AlertModel.find(q).sort({ timestamp: -1 }).limit(filters.limit || 50);
}
```



Filtered Alerts with severity as 'Warning'



Filtered Alerts with severity ‘All’

12. Main Alert Routes

Routes control how we interact with alerts.

Code Snippet:

```
// backend/src/routes/alerts.ts (excerpt)
router.get('/', authMiddleware(), alertController.listAlerts);
router.post('/:id/resolve', authMiddleware(), alertController.resolveAlert);
router.get('/:id/history', authMiddleware(), /* history controller */);
router.post('/rules/reload', authMiddleware(['admin']), async (req, res) => {
  /* reload rules */
});
```

13. Frontend Alert Refresh and Filter

The frontend keeps showing the latest alerts. It refreshes every 8 seconds without losing the current severity filter because we store it in a ref.

Code Snippet:

```
// client/src/App.tsx (excerpt)
const [severityFilter, setSeverityFilter] = useState<'ALL' | string>('ALL');
const severityFilterRef = useRef(severityFilter);

useEffect(() => {
  severityFilterRef.current = severityFilter;
}, [severityFilter]);

async function loadAlerts() {
  const current = severityFilterRef.current;
  const list = await fetchAlerts({
    limit: 50,
    severity: current === 'ALL' ? undefined : current
  });
  setAlerts(list);
}

useEffect(() => {
  if (!token) return;
  loadAlerts();
  const id = setInterval(loadAlerts, 8000);
  return () => clearInterval(id);
```

}, [token]);

14. Rule Engine

Rules are loaded at startup from rules.json. They can do three main things:

1. Set the initial severity when an event first becomes an alert.
2. Escalate existing alerts if many similar ones happen in a short window.
3. Auto-close alerts if certain metadata flags show a condition is now OK.

14.1 Rules File Format (New Format)

The format supports arrays and separate sections:

Code Snippet (JSON):

```
{
  "rules": [
    {
      "ruleId": "speed_high",
      "eventTypes": ["driver"],
      "severity": "CRITICAL",
      "condition": {
        "speed": { "$gt": 120 }
      }
    }
  ],
  "escalation": {
    "driver": {
      "escalate_if_count": 5,
      "window_mins": 10,
      "escalate_to": "CRITICAL"
    }
  },
  "auto_close": {
    "document": {
      "auto_close_if": "document_valid",
      "check_field": "document_valid"
    }
  }
}
```

14.2 Loading and Hot Reload

Code Snippet (load):

```
// backend/src/services/ruleEngine.ts (excerpt)
export async function loadRules(): Promise<Rules> {
  const stats = await fs.stat(RULES_PATH);
  lastModifiedTime = stats.mtimeMs;
  const raw = await fs.readFile(RULES_PATH, 'utf-8');
  const parsed = JSON.parse(raw);

  if (parsed.rules && Array.isArray(parsed.rules)) {
    rulesCache = {
      rules: parsed.rules,
      escalation: parsed.escalation || {},
      auto_close: parsed.auto_close || {}
    };
  } else {
    // legacy sanitize logic
  }
  return rulesCache;
}
```

The function checkAndReloadRules() compares last modified time to support hot reload while the app is running.

14.3 Matching an Incoming Event

When an event arrives the function evaluateEventAgainstRules(event) runs:

1. Reload rules if file changed.
2. Filter rules by matching eventTypes vs event.sourceType.
3. For each matching rule, evaluate its condition against the event metadata.
4. The first condition that passes decides the severity.
5. If none match, use default severity (or INFO).

Code Snippet (evaluate severity):

```
const matchingRules = rules.filter((rule: any) =>
  rule.eventTypes && rule.eventTypes.includes(event.sourceType)
);

for (const rule of matchingRules) {
  if (evaluateCondition(event.metadata, rule.condition)) {
    return { severity: rule.severity, matchedRule: rule };
  }
```

```
}
return { severity: event.severity || 'INFO' };
```

14.4 Condition Evaluation

We support simple MongoDB-like operators inside the rule condition: $eq, $gt, $gte, $lt, $lte, $ne.

Code Snippet (operators loop simplified):

```
for (const [op, expected] of Object.entries(operators as any)) {
  switch (op) {
    case '$gt': if (value <= expected) return false; break;
    case '$lt': if (value >= expected) return false; break;
    // ... others similar
  }
}
```

This keeps logic flexible without writing custom code for each field.

14.5 Escalation Logic (After Creation)

After an alert is created, evaluateOnCreate(alert) checks escalation rules:

1. Find escalation rule by alert.sourceType.
2. If it sets escalate_if_count and window_mins, count how many alerts of the same type (and maybe same driver) happened in the time window.
3. If count threshold is met, mark existing alerts as ESCALATED and optionally bump severity to escalate_to.

Code Snippet (threshold check):

```
const count = await alertsService.countAlertsInWindow(keyFilter, windowStart, windowEnd);
if (count >= rule.escalate_if_count) {
  // escalate all matching docs
}
```

14.6 Auto-Close Logic

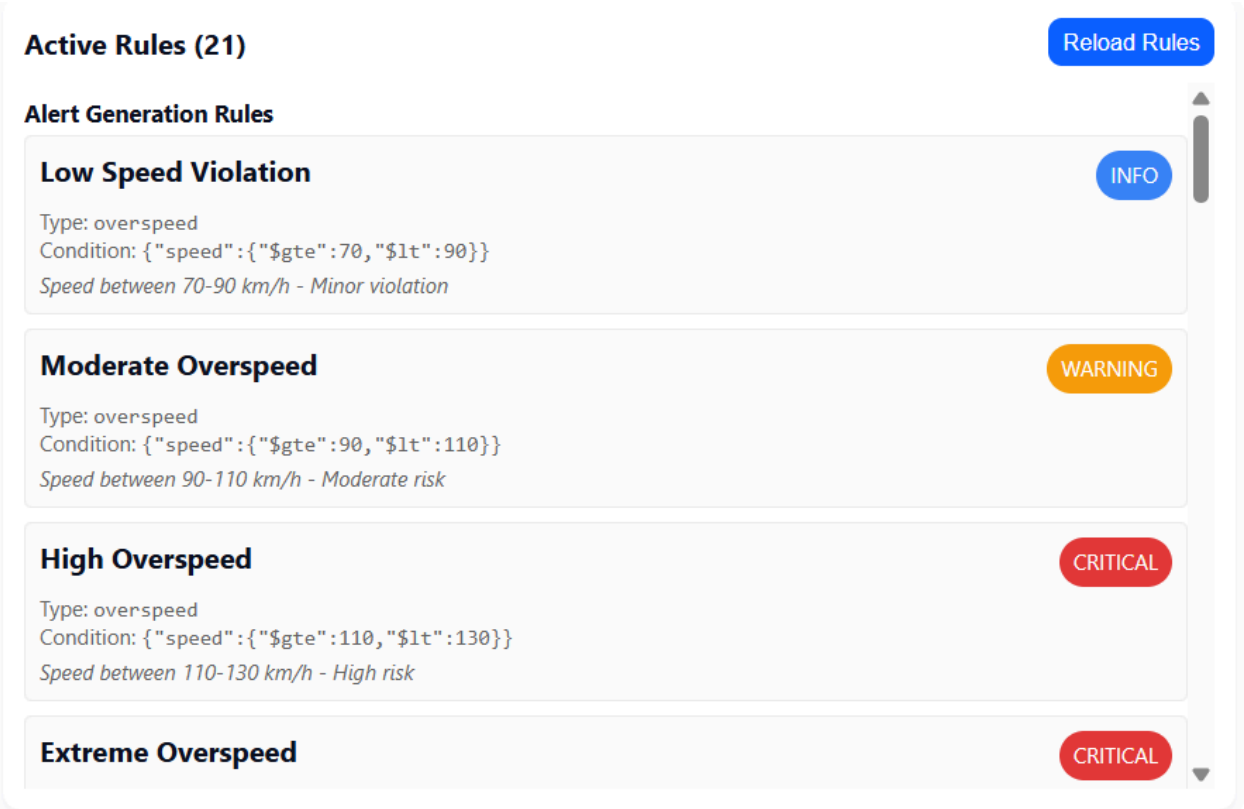
If the auto_close section defines a rule for the source type, we look for a metadata flag (like document_valid or document_renewed). If true, we update the alert status to AUTO-CLOSED and push a history record.

Code Snippet (auto close excerpt):

```
const shouldAutoClose = alert.metadata?.[key] === true ||
                        alert.metadata?.document_valid === true ||
                        alert.metadata?.document_renewed === true;

if (shouldAutoClose) {
  await AlertModel.updateOne(
    { alertId: alert.alertId },
    { $set: { status: 'AUTO-CLOSED' }, $push: { history: { state: 'AUTO-CLOSED' }}}
  );
}
```

14.7 Summary of Rule Engine Flow

1. Load and cache rules from file.
2. For each incoming event: match rules, assign severity.
3. On alert creation: check escalation and auto-close.
4. Workers also re-evaluates alerts over time.



Rules panel on dashboard

15. Worker Process (Background Tasks)

The worker runs every 2 minutes by default using a cron expression (*/2 * * * *). It does two main things:

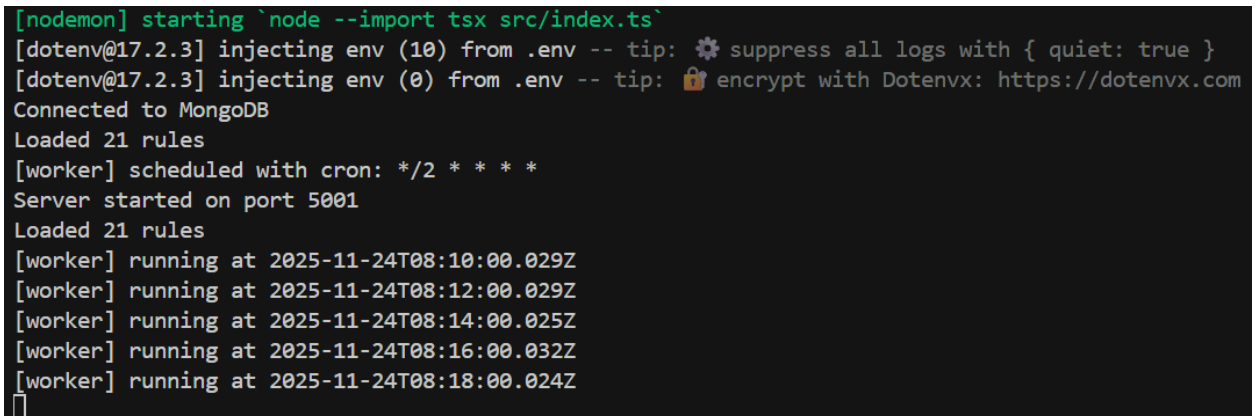
1. Auto-close alerts that are older than an expiry window (configured hours) and still OPEN/ESCALATED.
2. Re-run rule evaluation (evaluateAlert) to catch escalation or auto-close triggers set after initial creation.

Code Snippet (cron setup):

```
// backend/src/services/worker.ts (excerpt)
const expression = config.workerCron || '*/2 * * * *';
task = cron.schedule(expression, async () => {
  await processPendingAlerts();
});
```

Code Snippet (time-based auto close excerpt):

```
if (now - a.timestamp.getTime() >= expiryMs) {
  await AlertModel.updateOne({ alertId: a.alertId }, {
    $set: { status: 'AUTO-CLOSED', lastTransitionReason: 'TIME_WINDOW_EXPIRED' },
    $push: { history: { state: 'AUTO-CLOSED', reason: 'TIME_WINDOW_EXPIRED' }}
  });
}
```



Worker running every two minutes

16. Event Logging and History

Every change or significant action logs an event document. This keeps a timeline.

16.1 Event Types

CREATED, ESCALATED, AUTO_CLOSED, RESOLVED, INFO (misc informational events, like severity bump).

16.2 Logging Function

Code Snippet:

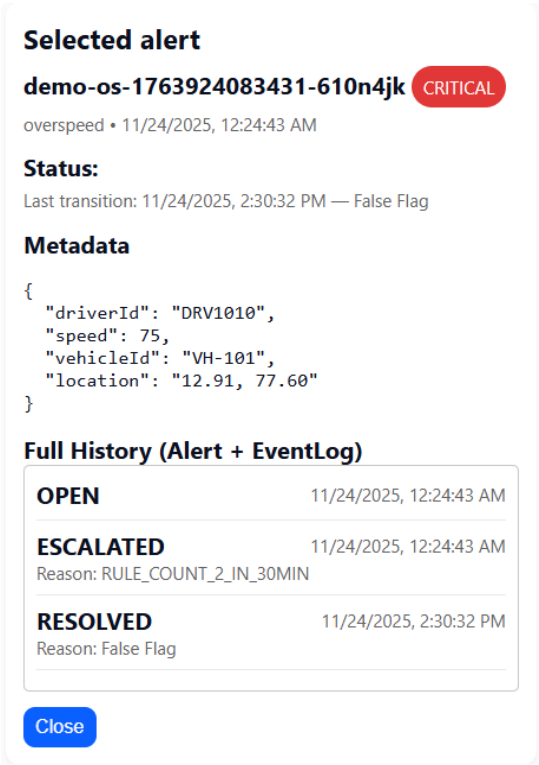
```
// backend/src/services/eventService.ts (excerpt)
export async function logEvent(payload: {
  alertId: string;
  type: 'CREATED' | 'ESCALATED' | 'AUTO_CLOSED' | 'RESOLVED' | 'INFO';
}) {
  const doc = await EventLogModel.create({
    alertId: payload.alertId,
    type: payload.type,
    ts: new Date(),
    payload: payload.payload || {}
  });

  emitEvent({
    alertId: doc.alertId,
    type: doc.type,
    ts: doc.ts.toISOString(),
    payload: doc.payload
  });

  return doc;
}
```

16.3 Why This Helps

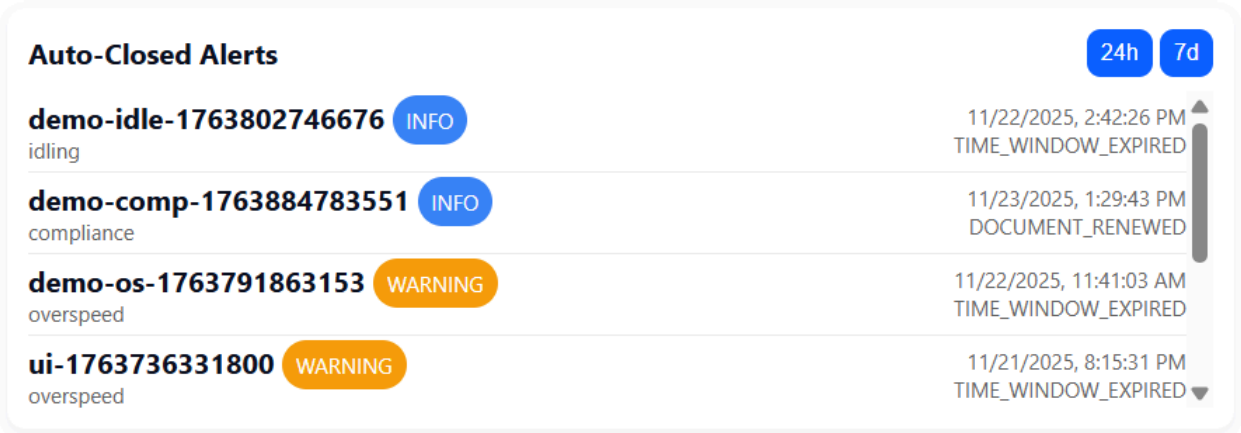
It separates lifecycle history from the alert itself. We can aggregate and analyze events without touching the main alert document. It also supports streaming (via emitEvent).

17. Putting It All Together (Lifecycle Example)

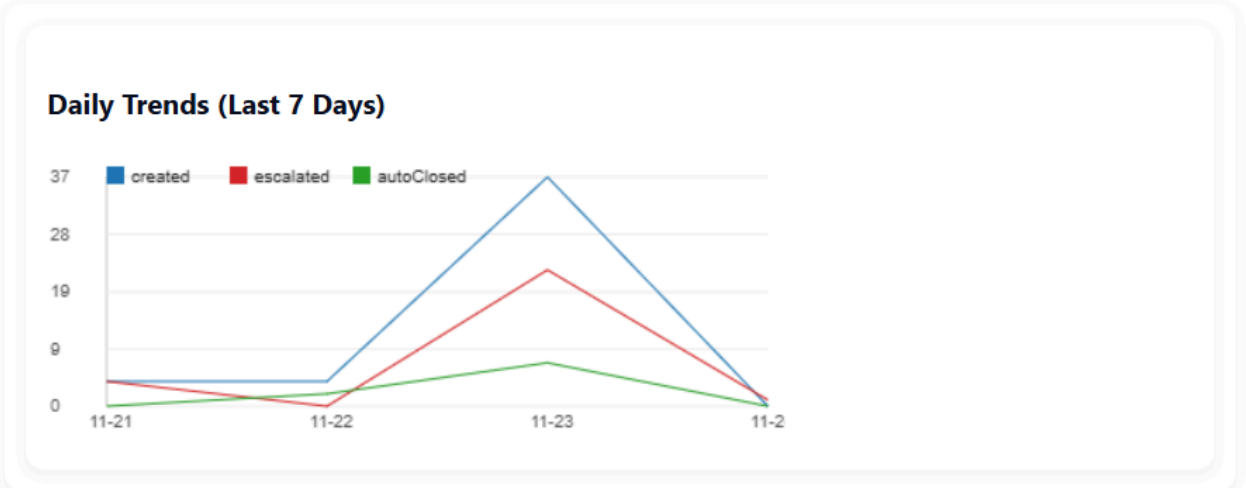
1. **Event arrives with metadata:** { speed: 130, driverId: 'D12' }.
2. **Rule engine:** Sees driver event type, matches speed_high rule (speed > 120). Severity becomes CRITICAL.
3. **Alert saved:** With OPEN status, history has one record.
4. **Worker runs later:** Sees 6 similar alerts for same driver in 10 min window. Escalation rule triggers: alerts marked ESCALATED and severity bumped if configured.
5. **Driver updates document:** Metadata now has document_valid: true. Auto-close rule triggers, status becomes AUTO-CLOSED.
6. **User views history timeline:** CREATED -> ESCALATED -> AUTO-CLOSED.



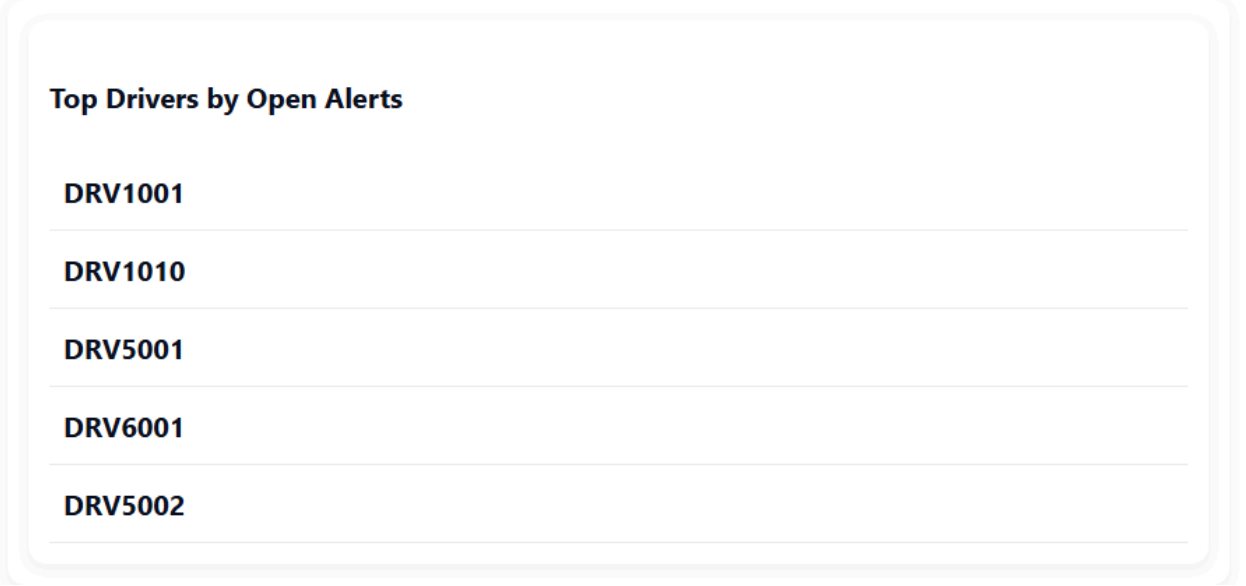
View information about a singular alert



Auto-closed alerts in last 7 days



Trends of numbers of created, escalated and auto-closed alerts



List of 5 drivers with most open alerts currently

UI to create demo events by using predefined events in drop down or editing payload

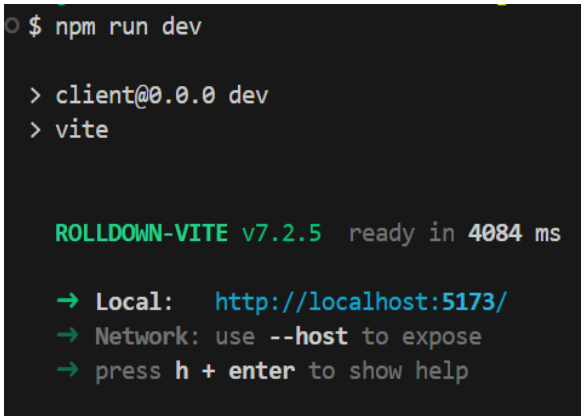
18. Running the Project

Steps:

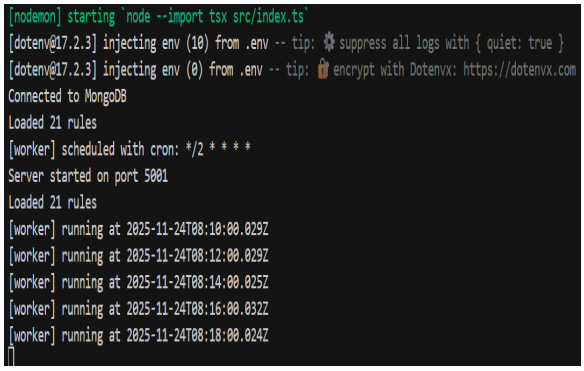
1. Clone repository.
2. Open terminal and run:
   cd backend
   npm install
   npm run dev

3. Open a second terminal:
   cd client
   npm install
   npm run dev

4. Make sure MongoDB is running or set a proper connection string in .env.
5. Open the frontend URL shown in the terminal (usually http://localhost:5173 or similar).
6. Log in to view the dashboard.



Running Frontend



Running Backend

19. Error Handling Basics

- **Backend:** Central error handler returns JSON with error message.
- **Frontend:** Shows a simple alert or console error if a request fails.
- **Auth:** If the token fails, it is cleared and the user is logged out.

20. Simple Security Measures

- **Helmet:** Adds helpful HTTP headers.
- **CORS:** Let's frontend talk to the backend safely.
- **Token:** Stored in sessionStorage (okay for demo, would harden later).

21. Challenges Faced

- **Auto-refresh** was first wiping severity filter. Fixed by using a ref to keep the latest value.
- **Joining alert history** and event log in a clean view.
- **Keeping UI simple** without heavy styling libraries.

22. Possible Future Improvements

- Pagination and more filters (date range, status).
- Stronger auth with refresh tokens and password resets.
- Visual rule editor (create/edit rules in browser).
- WebSocket live updates instead of polling.

- More polished responsive design for mobile.
- Alert grouping or correlation for related alerts.

23. Conclusion

The **Intelligent Alert System** successfully demonstrates a foundational pattern for transforming raw, continuous event data into manageable, actionable alerts. By integrating a straightforward Node.js/Express backend with MongoDB and a React frontend, the project delivers on its core objectives: automated severity assignment via a simple rule engine, real-time filtering, historical tracking of alert lifecycles, and essential dashboard metrics.

This system is more than just an event viewer; it is a proof-of-concept for intelligent data filtering, a critical requirement in real-world operational environments. The modular architecture, clear data model, and defined service layers ensure that the solution is robust, maintainable, and highly extensible for future features like advanced correlation, more complex rule logic, and enhanced security measures. The project stands as a solid base for further development into a full-scale monitoring platform.