National University of Computer and Emerging Sciences
Islamabad Campus

Semester Project
AI-Powered Secure SDLC Compliance Auditor

Secure Software Design

Submitted by: Mustan Sir Hussain, Muazam Ali , Tauqeer Ahmed , Saad Mehmood
Roll number: i221764 , i221734 ,i221655 , 22i-2078)
Date: 29-Nov-2025

1. Objectives and Problem Statement

Problem Statement

Modern software development faces critical security challenges as vulnerabilities in code lead to data breaches, financial losses, and compromised systems. Traditional security testing occurs late in the Software Development Life Cycle (SDLC), making fixes expensive and time-consuming. Developers lack integrated tools that provide:

- Real-time security analysis
- Automated remediation
- Comprehensive threat modeling during early development

These limitations increase the likelihood of insecure applications, compliance issues, and higher development costs.

Project Objectives

This project aims to develop an AI-powered security compliance auditor that:

- Integrates multiple static analysis tools (Semgrep, Bandit, ESLint, Flawfinder, Cppcheck) for multi-language vulnerability detection across Python, JavaScript, C++, and Java
- Provides AI-driven security insights using Groq's Llama 3.3 70B model for intelligent code analysis and automated fix generation
- Implements multi-framework threat modeling (STRIDE, PASTA, DREAD) for structured risk assessment
- Generates professional security reports for compliance and audit purposes
- Ensures secure authentication using JWT and bcrypt to protect user data and analysis history

2. Proposed Solution & Architecture

System Overview

The AI SDLC Security Compliance Auditor is a full-stack web platform built with:

- **React (Frontend)**
- **Node.js / Express (Backend)**
- **MongoDB (Database)**
- **Groq API with Llama 3.3 70B model** for AI-driven code security analysis
- **Static analysis tools** integrated for comprehensive vulnerability scanning

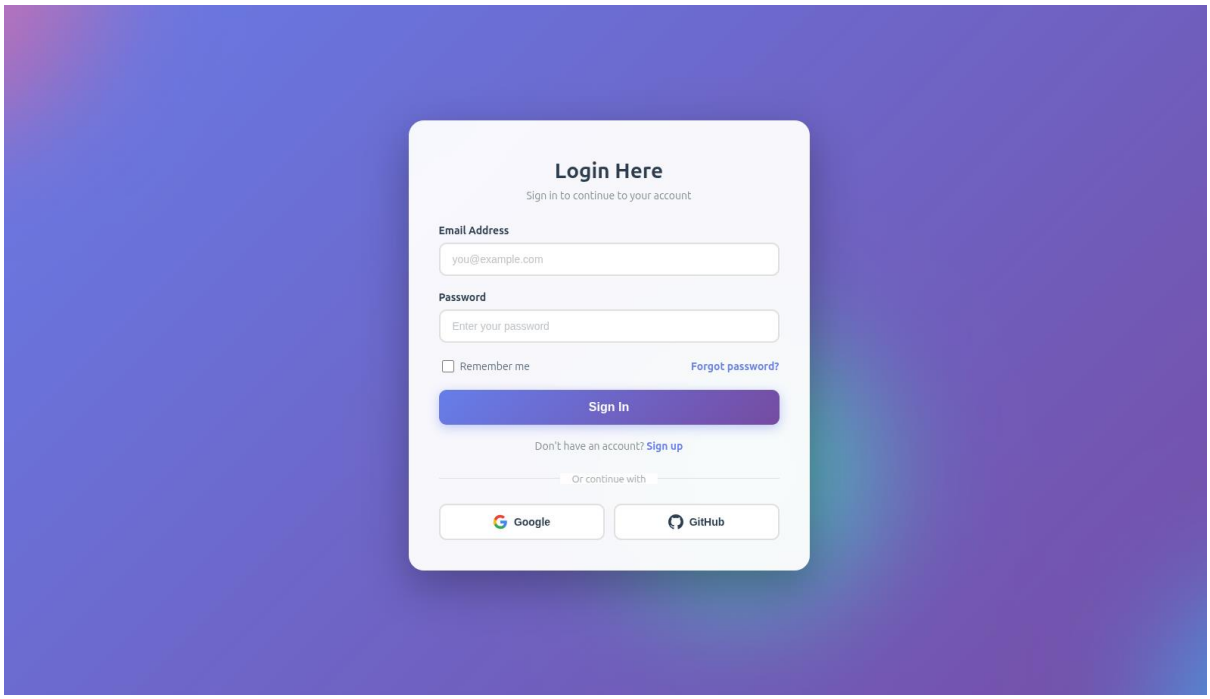
Architecture Diagram

2.3 Component Architecture

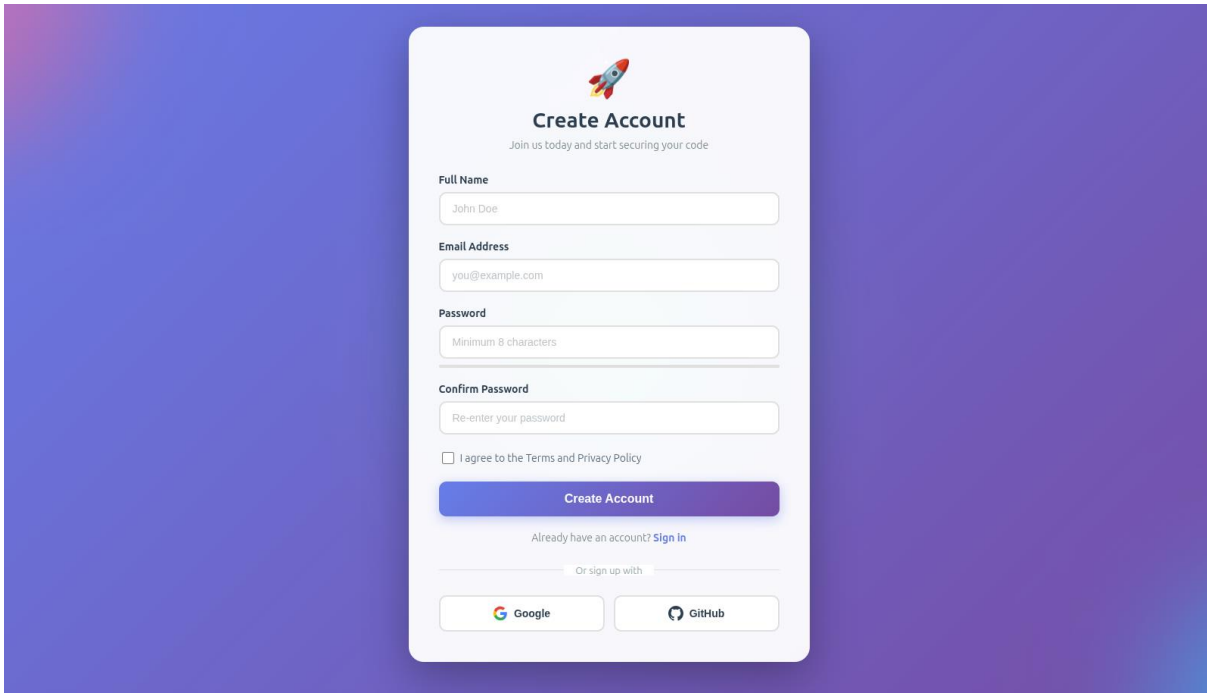
Frontend Components

- **Authentication Layer:** Login.js, Signup.js with form validation and password strength indicators
- **Analysis Interface:** MainApp.js with CodeMirror integration, multi-language support, real-time results
- **Remediation Module:** Fix.js with side-by-side code comparison and AI-generated fixes
- **Threat Modeling:** ThreatModel.js with framework selection and detailed analysis display
- **Routing:** React Router DOM for SPA navigation with protected routes

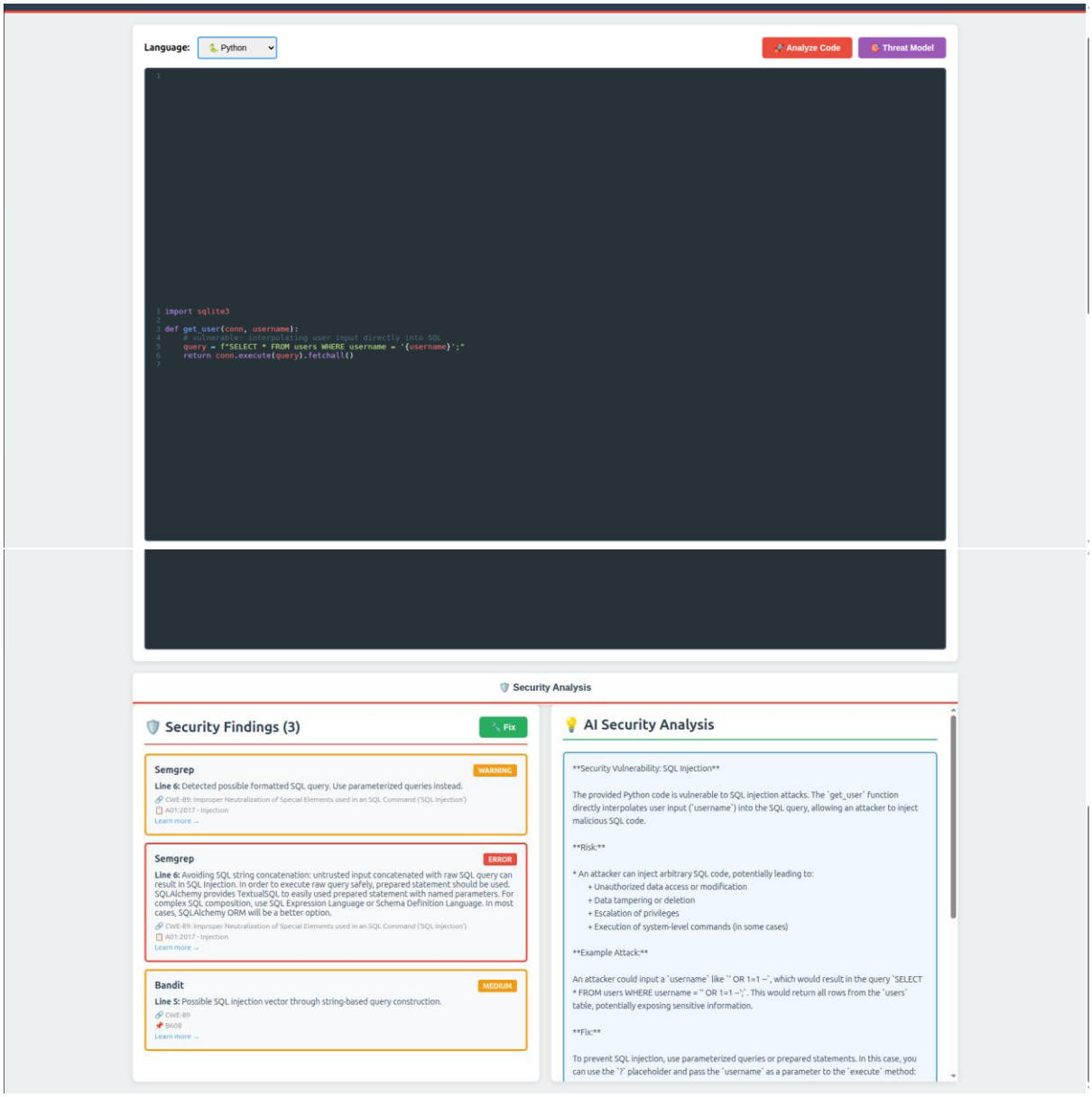
Login Page :

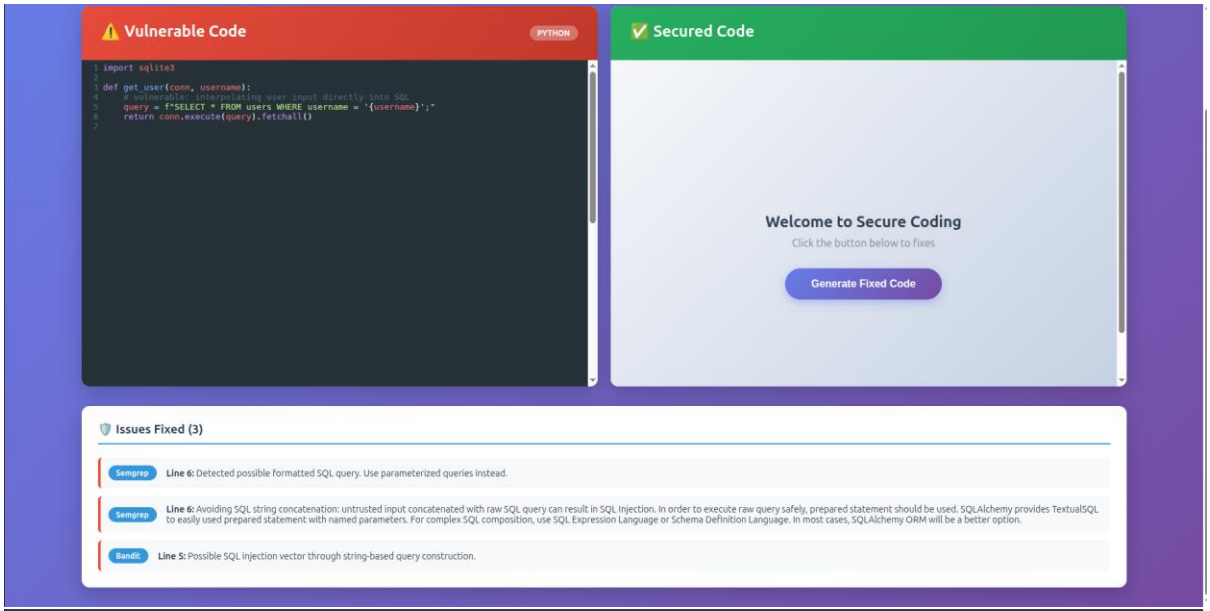


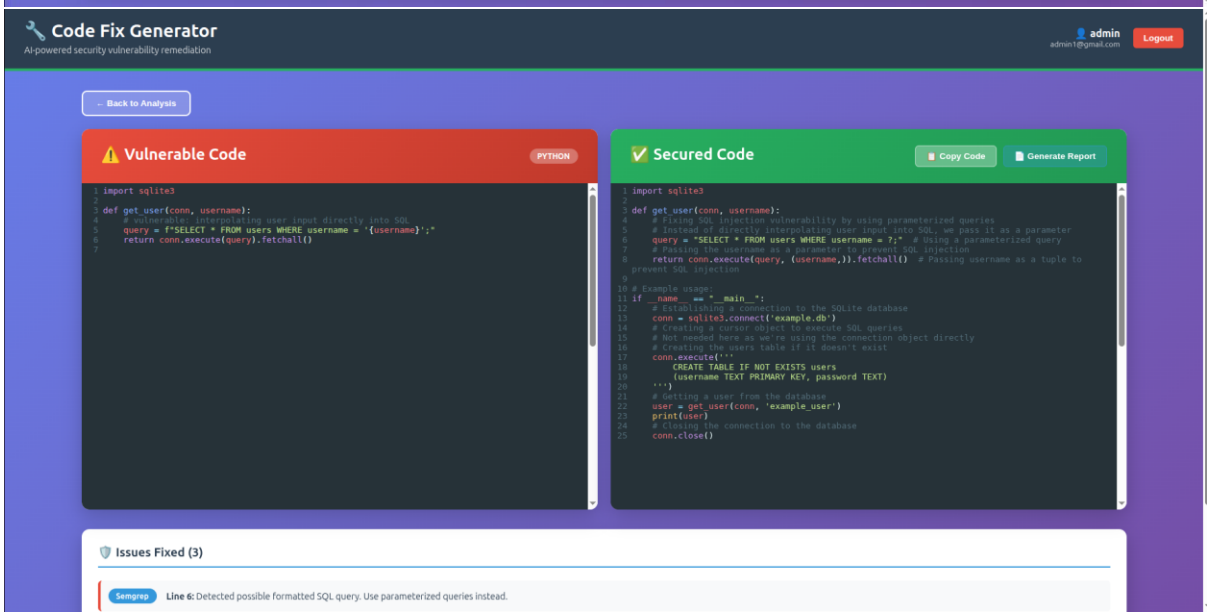
Sign up page:

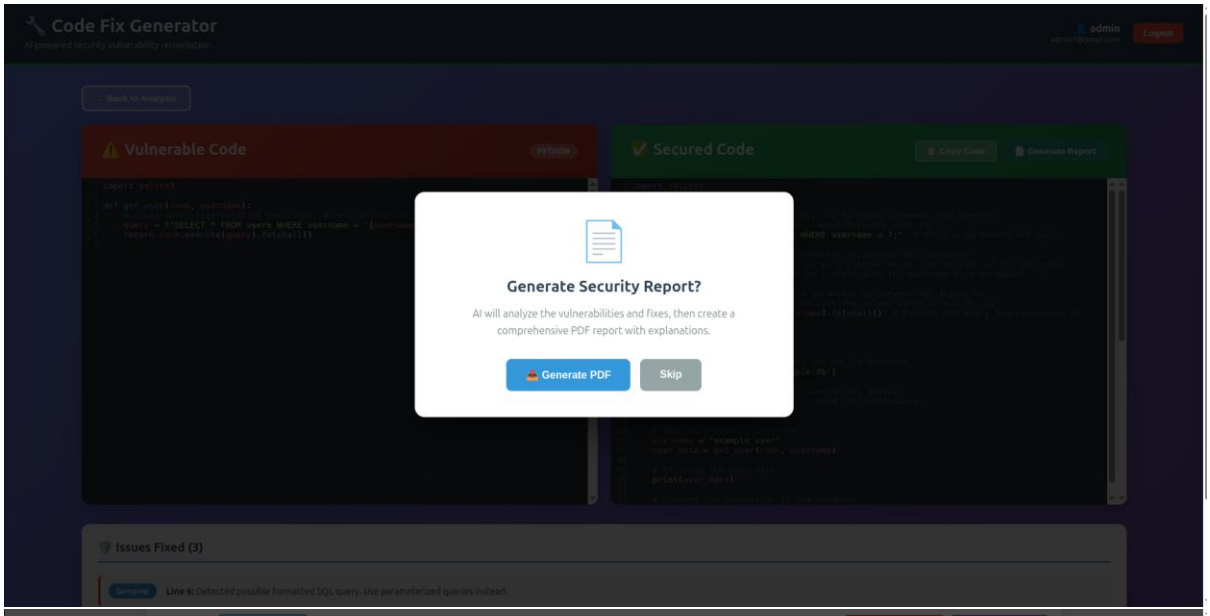

Create Account
Join us today and start securing your code
Full Name
John Doe
Email Address
you@example.com
Password
Minimum 8 characters
Confirm Password
Re-enter your password
I agree to the Terms and Privacy Policy
Create Account
Already have an account? Sign in
Or sign up with
Google
GitHub

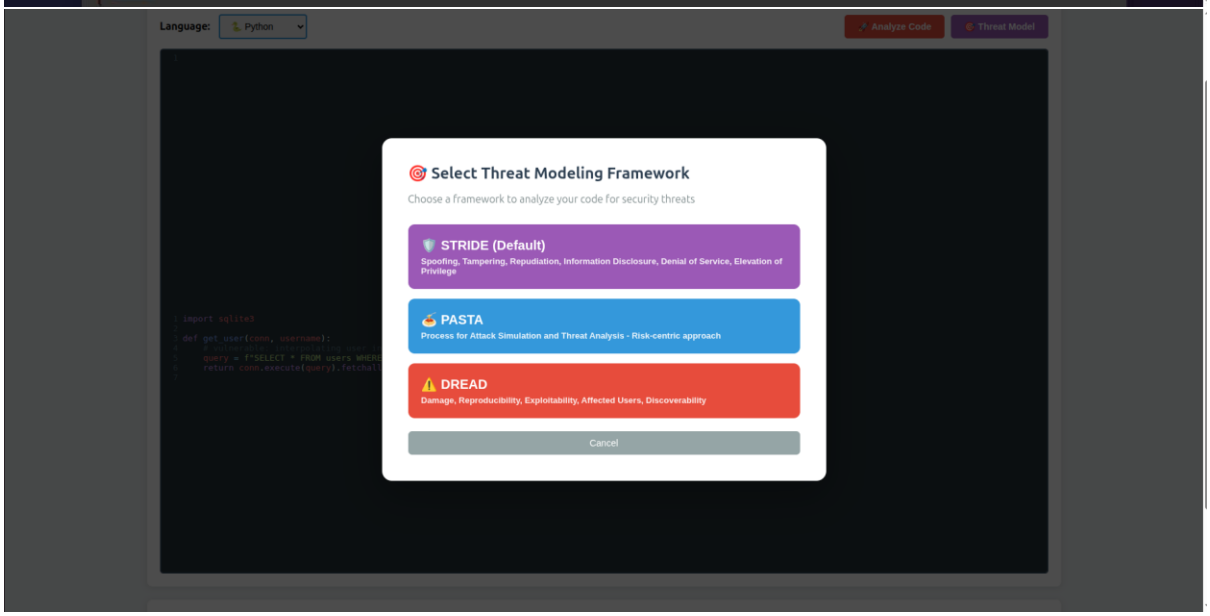
Display :

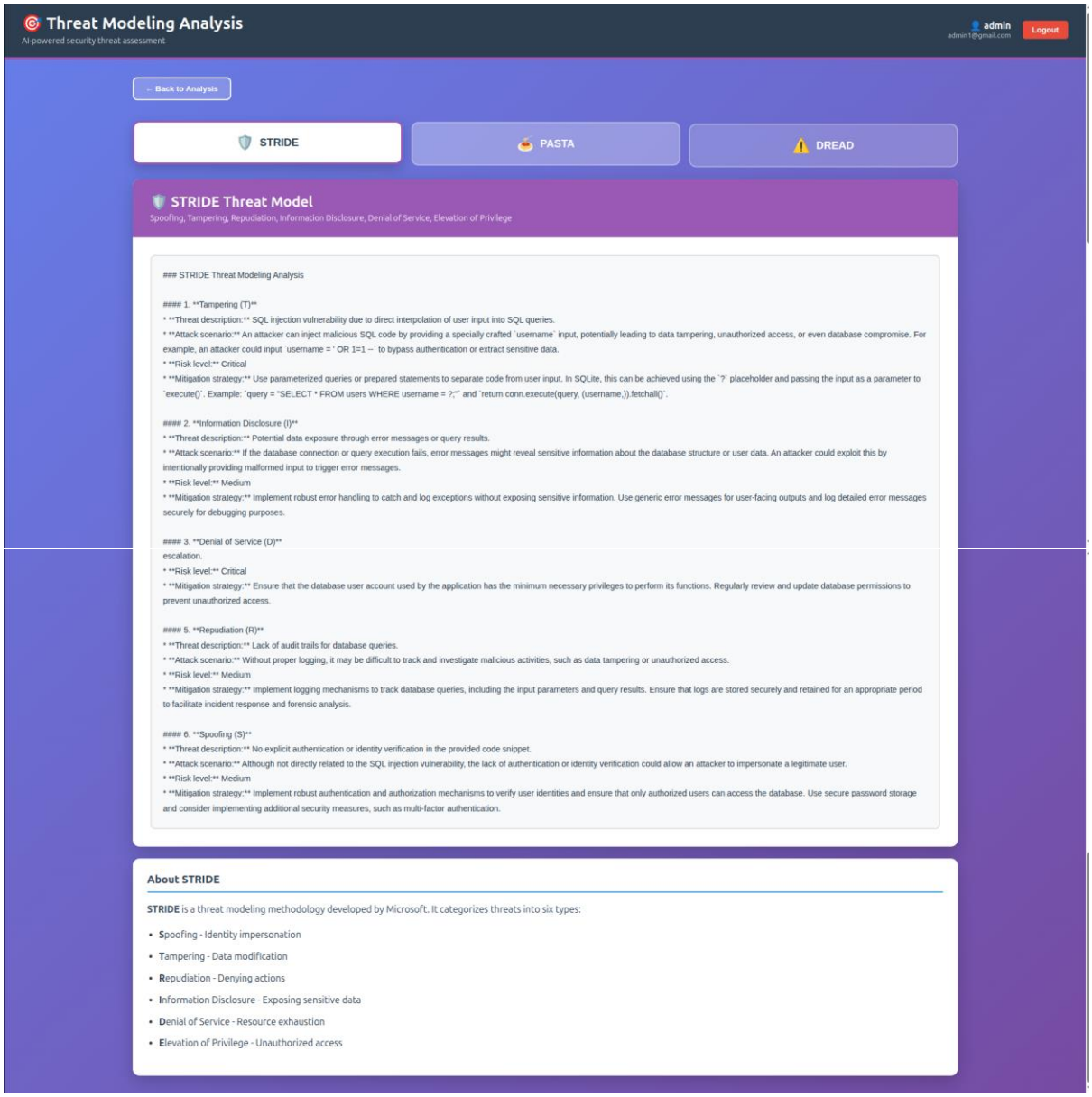

Language:
Python
Analyze Code
Threat Model
import sqlite3
def get_user(conn, username):
# vulnerable: interpolating user input directly into SQL
query = f"SELECT * FROM users WHERE username = '{username}';"
return conn.execute(query).fetchall()
Security Analysis
Security Findings (3)
Fix
Semgrep
WARNING
Line 6: Detected possible formatted SQL query. Use parameterized queries instead.
CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
A01:2017 - Injection
Learn more ...
Semgrep
ERROR
Line 6: Avoiding SQL string concatenation: untrusted input concatenated with raw SQL query can result in SQL injection. In order to execute raw query safely, prepared statement should be used. SQLAlchemy provides TextualSQL to easily used prepared statement with named parameters. For complex SQL composition, use SQL Expression Language or Schema Definition Language. In most cases, SQLAlchemy ORM will be a better option.
CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
A01:2017 - Injection
Learn more ...
Bandit
MEDIUM
Line 5: Possible SQL injection vector through string-based query construction.
CWE-89
B608
Learn more ...
AI Security Analysis
Security Vulnerability: SQL Injection
The provided Python code is vulnerable to SQL injection attacks. The `get_user` function directly interpolates user input (`username`) into the SQL query, allowing an attacker to inject malicious SQL code.
Risk:
* An attacker can inject arbitrary SQL code, potentially leading to:
+ Unauthorized data access or modification
+ Data tampering or deletion
+ Escalation of privileges
+ Execution of system-level commands (in some cases)
Example Attack:
An attacker could input a `username` like `' OR 1=1 --`, which would result in the query `SELECT * FROM users WHERE username = '' OR 1=1 --';`. This would return all rows from the `users` table, potentially exposing sensitive information.
Fix:
To prevent SQL injection, use parameterized queries or prepared statements. In this case, you can use the `?` placeholder and pass the `username` as a parameter to the `execute` method:


Vulnerable Code
PYTHON
import sqlite3
def get_user(conn, username):
# Vulnerable: interpolating user input directly into SQL
query = f"SELECT * FROM users WHERE username = '{username}';"
return conn.execute(query).fetchall()
Secured Code
Welcome to Secure Coding
Click the button below to fixes
Generate Fixed Code
Issues Fixed (3)
Semgrep
Line 6: Detected possible formatted SQL query. Use parameterized queries instead.
Semgrep
Line 6: Avoiding SQL string concatenation: untrusted input concatenated with raw SQL query can result in SQL Injection. In order to execute raw query safely, prepared statement should be used. SQLAlchemy provides TextualSQL to easily used prepared statement with named parameters. For complex SQL composition, use SQL Expression Language or Schema Definition Language. In most cases, SQLAlchemy ORM will be a better option.
Bandit
Line 5: Possible SQL injection vector through string-based query construction.


Code Fix Generator
AI-powered security vulnerability remediation
admin
admin1@gmail.com
Logout
← Back to Analysis
Vulnerable Code
PYTHON
Secured Code
Copy Code
Generate Report
import sqlite3
def get_user(conn, username):
# Vulnerable: interpolating user input directly into SQL
query = f"SELECT * FROM users WHERE username = '{username}';"
return conn.execute(query).fetchall()
import sqlite3
def get_user(conn, username):
# Fixing SQL injection vulnerability by using parameterized queries
# Instead of directly interpolating user input into SQL, we pass it as a parameter
query = "SELECT * FROM users WHERE username = ?;" # Using a parameterized query
# Passing the username as a parameter to prevent SQL injection
return conn.execute(query, (username,)).fetchall() # Passing username as a tuple to prevent SQL injection
# Example usage
if __name__ == "__main__":
# Establishing a connection to the SQLite database
conn = sqlite3.connect('example.db')
# Creating a cursor object to execute SQL queries
# Not needed here as we're using the connection object directly
# Creating the users table if it doesn't exist
conn.execute('''
CREATE TABLE IF NOT EXISTS users
(username TEXT PRIMARY KEY, password TEXT)
''')
# Getting a user from the database
user = get_user(conn, 'example_user')
print(user)
# Closing the connection to the database
conn.close()
Issues Fixed (3)
Semgrep
Line 6: Detected possible formatted SQL query. Use parameterized queries instead.


Code Fix Generator
Vulnerable Code
Secured Code
Generate Security Report?
AI will analyze the vulnerabilities and fixes, then create a comprehensive PDF report with explanations.
Generate PDF
Skip
Issues Fixed (3)


Language:
Python
Analyze Code
Threat Model
Select Threat Modeling Framework
Choose a framework to analyze your code for security threats
STRIDE (Default)
Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Privilege
PASTA
Process for Attack Simulation and Threat Analysis - Risk-centric approach
DREAD
Damage, Reproducibility, Exploitability, Affected Users, Discoverability
Cancel



Backend Services

- **Authentication Service:** JWT token generation/verification, bcrypt password hashing (10 rounds)
- **Analysis Engine:** Parallel execution of security tools, temporary file management, result aggregation
- **AI Integration Service:** Groq API communication with timeout handling and error recovery
- **Database Service:** Mongoose ODM for user management and session handling

Data Flow

1. **User authentication** → JWT token stored in localStorage
2. **Code submission** → Temporary file creation (input.py, input.js, input.cpp, input.java)
3. **Parallel tool execution** → Semgrep + language-specific analyzers
4. **AI processing** → Groq API with security-focused prompts
5. **Result aggregation** → Combined JSON response with severity classification
6. **Code fixing** → AI generates secure alternatives with explanations
7. **Threat modeling** → Framework-specific analysis with risk assessment
8. **Report generation** → PDF creation with jsPDF and AI-written content

3. Methodology & SDLC Coverage

3.1 Development Approach

The project follows **Agile methodology** with **security-first design principles** and iterative feature development across seven phases.

3.2 SDLC Security Integration

Phase 1: Requirements Analysis

Security Activities:

- Defined security-critical features (authentication, code analysis, data protection)
- Identified compliance requirements (secure password storage, API security)
- Threat surface mapping (user input, file handling, external API calls)
- Selected security tools based on language coverage and vulnerability detection capabilities

Deliverables:

- Security requirements specification
- Tool selection matrix (Semgrep, Bandit, ESLint, Flawfinder, Cppcheck)
- API security requirements (CORS, rate limiting, authentication)

Phase 2: Secure Design

Security Activities:

- Architecture design with separation of concerns (frontend/backend/database)
- Threat modeling during design (authentication bypass, injection attacks, session hijacking)
- Selection of secure frameworks (JWT for tokens, bcrypt for passwords, Mongoose for MongoDB)
- API endpoint design with authentication requirements
- Database schema design with validation rules

Deliverables:

- System architecture diagram
- Database schema with security constraints (unique email, required fields)
- API specification with authentication headers
- Security control mapping (authentication, authorization, input validation)

Phase 3: Implementation (Secure Coding)

Security Activities:

- **Authentication:** JWT implementation with 7-day expiry, bcrypt with 10 salt rounds
- **Input Validation:** Schema validation with Mongoose, email format validation, password strength requirements
- **Secure Configuration:** Environment variables for secrets, .gitignore for sensitive files
- **Error Handling:** Generic error messages to prevent information disclosure
- **Dependency Management:** Regular updates, vulnerability scanning with npm audit
- **Code Review:** Security-focused code review for all authentication and data handling logic

Security Best Practices Applied:

```
// Password Hashing
const salt = await bcrypt.genSalt(10);
const hashedPassword = await bcrypt.hash(password, salt);

// JWT Token Generation
const token = jwt.sign({ userId: user._id }, JWT_SECRET, { expiresIn: "7d" });

// Input Validation
const userSchema = new mongoose.Schema({
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true }
});

// Secure API Communication
headers: { "Authorization": `Bearer ${API_KEY}` }
```

Phase 4: Testing & Validation

Security Testing Performed:

- **Authentication Testing:** Login/logout, token expiry, password validation, session management
- **Input Validation Testing:** SQL injection attempts, XSS payloads, command injection
- **API Security Testing:** Unauthorized access attempts, CORS verification, rate limiting
- **SAST Implementation:** Semgrep, Bandit, ESLint on project codebase
- **Functional Testing:** All features tested with valid and malicious inputs

Test Cases:

- **SQL Injection (Python):**

query = "SELECT * FROM users WHERE username = '" + username + "'"

- **XSS (JavaScript):**

document.getElementById('output').innerHTML = userInput;

- **Buffer Overflow (C++):** Unbounded strcpy operations
- **Hardcoded Credentials:** API keys stored in source code

Phase 5: Deployment & Maintenance

Security Activities:

- Docker containerization for MongoDB with authentication enabled
- Environment variable configuration for production secrets
- CORS configuration for allowed origins
- HTTPS readiness (development on HTTP; production recommendations included)
- Dependency updates and vulnerability patching
- Monitoring and logging setup

4. Threat Modeling & Risk Analysis

4.1 Threat Identification

4.1.1 STRIDE Analysis

Spoofing

- **Threat:** Attacker impersonates legitimate user through stolen credentials
- **Mitigation:** bcrypt password hashing (10 rounds), JWT tokens with expiry, secure session management
- **Implementation:** Login verification checks password hash, tokens expire after 7 days

Tampering

- **Threat:** Modification of code analysis results during transmission
- **Mitigation:** HTTPS in production, signed JWT tokens prevent tampering
- **Implementation:** JWT signature verification on every protected route

Repudiation

- **Threat:** Users deny performing security analysis or accessing reports
- **Mitigation:** User association with MongoDB records, JWT tokens link actions to users
- **Implementation:** User ID embedded in JWT payload, can be extended with audit logging

Information Disclosure

- **Threat:** Exposure of source code, API keys, or user credentials
- **Mitigation:** Environment variables for secrets, .gitignore for sensitive files, no code storage

- **Implementation:** Temporary files deleted after analysis, API keys in .env, passwords hashed

Denial of Service

- **Threat:** Resource exhaustion through large code submissions or repeated requests
- **Mitigation:** API timeouts (30–50 seconds), Groq rate limiting, file size validation
- **Implementation:** Axios timeout configuration, temporary file cleanup

Elevation of Privilege

- **Threat:** Unauthorized access to admin functions or other users' data
- **Mitigation:** JWT verification middleware, user-specific data queries, no privilege escalation paths
- **Implementation:** verifyToken middleware on protected routes, MongoDB queries filtered by user ID

4.1.2 Attack Surface Analysis

External Attack Vectors:

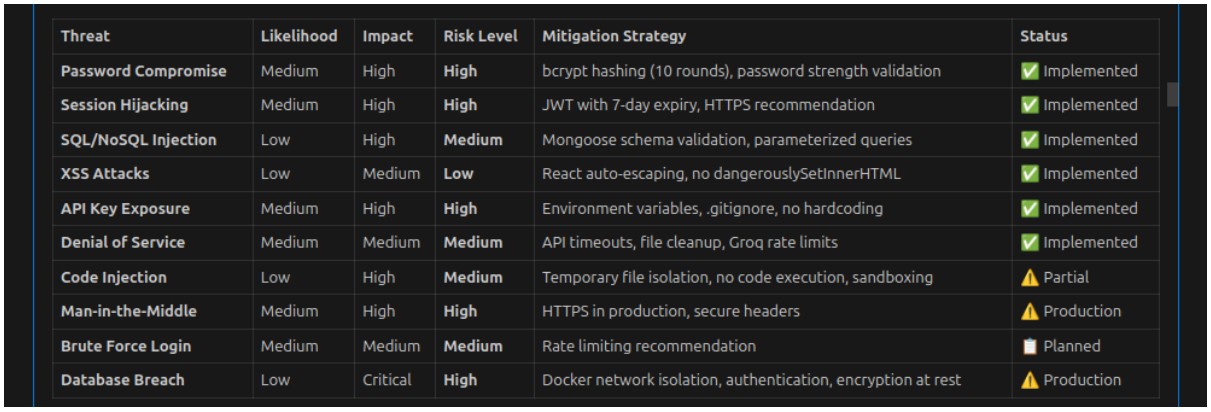
- Login/Signup forms (credential stuffing, brute force)
- Code editor input (code injection, malicious payloads)
- API endpoints (unauthorized access, parameter manipulation)
- MongoDB connection (authentication bypass, NoSQL injection)
- External API dependencies (Groq API compromise)

Internal Attack Vectors:

- JWT token theft (XSS, man-in-the-middle)
- Session hijacking (token reuse)
- Database credential exposure (.env file access)

4.2 Risk Assessment & Mitigation

Table 1: Risk Assessment Matrix



Threat	Likelihood	Impact	Risk Level	Mitigation Strategy	Status
Password Compromise	Medium	High	**High**	bcrypt hashing (10 rounds), password strength validation	✅ Implemented
Session Hijacking	Medium	High	**High**	JWT with 7-day expiry, HTTPS recommendation	✅ Implemented
SQL/NoSQL Injection	Low	High	**Medium**	Mongoose schema validation, parameterized queries	✅ Implemented
XSS Attacks	Low	Medium	**Low**	React auto-escaping, no dangerouslySetInnerHTML	✅ Implemented
API Key Exposure	Medium	High	**High**	Environment variables, .gitignore, no hardcoding	✅ Implemented
Denial of Service	Medium	Medium	**Medium**	API timeouts, file cleanup, Groq rate limits	✅ Implemented
Code Injection	Low	High	**Medium**	Temporary file isolation, no code execution, sandboxing	⚠️ Partial
Man-in-the-Middle	Medium	High	**High**	HTTPS in production, secure headers	⚠️ Production
Brute Force Login	Medium	Medium	**Medium**	Rate limiting recommendation	📋 Planned
Database Breach	Low	Critical	**High**	Docker network isolation, authentication, encryption at rest	⚠️ Production

Risk Prioritization

- **Critical (Production):** HTTPS, database encryption, JWT secret rotation
- **High (Implemented):** Password hashing, session management, API key protection
- **Medium (Partial):** Code sandboxing, rate limiting, advanced logging
- **Low (Monitoring):** Input sanitization, CSP headers, security headers

4.3 Mitigation Justification

Password Security (bcrypt):

- **Rationale:** 10 salt rounds provide strong protection against rainbow table and brute force attacks
- **Feasibility:** Native bcrypt.js library, minimal performance impact (~100ms per hash)
- **Trade-off:** Slight login delay acceptable for security gain

JWT Authentication:

- **Rationale:** Stateless authentication reduces database queries, enables scalability
- **Feasibility:** Industry-standard jsonwebtoken library, localStorage for client storage
- **Trade-off:** 7-day expiry balances security (shorter lifespan) and UX (less re-authentication)

MongoDB with Docker:

- **Rationale:** Containerization provides isolation, easy deployment, version control
- **Feasibility:** docker-compose setup, minimal configuration required

- **Trade-off:** Docker overhead acceptable for development consistency and production parity

Groq API (Free Tier):

- **Rationale:** No cost, powerful Llama 3.3 70B model, good rate limits for MVP
- **Feasibility:** Simple REST API, axios integration, comprehensive documentation
- **Trade-off:** Rate limits acceptable for educational project; can upgrade for production

Temporary File Strategy:

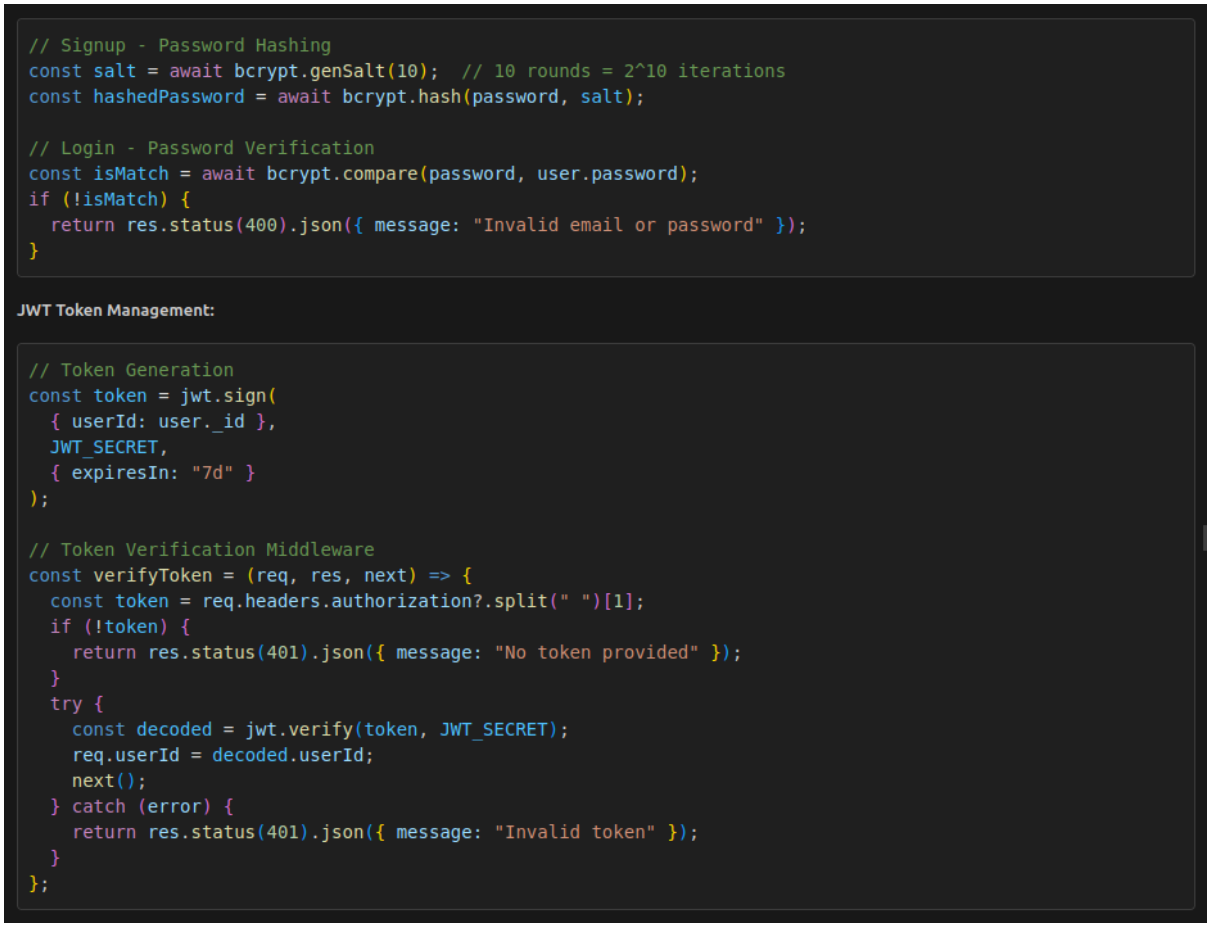
- **Rationale:** Security tools require file input, temporary files minimize storage risk
- **Feasibility:** fs.writeFileSync with synchronous cleanup
- **Trade-off:** Small disk I/O overhead acceptable for security tool compatibility

5. Code Implementation & Security Practices

5.1 Secure Coding & Best Practices

5.1.1 Authentication & Authorization

Password Hashing:



```
// Signup - Password Hashing
const salt = await bcrypt.genSalt(10);  // 10 rounds = 2^10 iterations
const hashedPassword = await bcrypt.hash(password, salt);

// Login - Password Verification
const isMatch = await bcrypt.compare(password, user.password);
if (!isMatch) {
  return res.status(400).json({ message: "Invalid email or password" });
}
```

JWT Token Management:

```
// Token Generation
const token = jwt.sign(
  { userId: user._id },
  JWT_SECRET,
  { expiresIn: "7d" }
);

// Token Verification Middleware
const verifyToken = (req, res, next) => {
  const token = req.headers.authorization?.split(" ")[1];
  if (!token) {
    return res.status(401).json({ message: "No token provided" });
  }
  try {
    const decoded = jwt.verify(token, JWT_SECRET);
    req.userId = decoded.userId;
    next();
  } catch (error) {
    return res.status(401).json({ message: "Invalid token" });
  }
};
```

5.1.2 Input Validation

Mongoose Schema Validation:

```
const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true },
  createdAt: { type: Date, default: Date.now }
});
```

Frontend Validation:

```
// Email validation
const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
if (!emailRegex.test(email)) {
  setError("Invalid email format");
  return;
}

// Password strength
const passwordStrength = password.length < 6 ? "weak" :
                         password.length < 10 ? "medium" : "strong";
```

5.1.3 Secure API Communication

Environment Variables:

```
dotenv.config();
const API_KEY = process.env.GROQ_API_KEY;
const MONGODB_URI = process.env.MONGODB_URI;
const JWT_SECRET = process.env.JWT_SECRET;
```

CORS Configuration:

```
app.use(cors({
  origin: 'http://localhost:3001',  // Specific origin in development
  credentials: true
}));
```

5.1.4 Error Handling

Generic Error Messages:

```
// Avoid: "User not found" vs "Incorrect password" (username enumeration)
return res.status(400).json({ message: "Invalid email or password" });

// Avoid exposing stack traces
catch (error) {
  console.error("Login error:", error);  // Server-side logging
  res.status(500).json({ message: "Server error during login" });  // Generic client message
}
```

5.2 Functionality & Correctness

Core Features Implemented

Multi-Language Security Analysis

- ☑ Python (Semgrep + Bandit)
- ☑ JavaScript (Semgrep + ESLint)
- ☑ C++ (Semgrep + Flawfinder + Cppcheck)
- ☑ Java (Semgrep)

AI-Powered Features

- ☑ Security analysis and vulnerability explanation
- ☑ Automated code fixing with secure alternatives
- ☑ Threat modeling (STRIDE, PASTA, DREAD)
- ☑ Professional report generation

User Management

- ☑ User registration with validation
- ☑ Secure login with JWT
- ☑ Session persistence
- ☑ Logout functionality

Analysis Workflow

- ☑ Code editor with syntax highlighting
- ☑ Language selection
- ☑ Real-time analysis with loading states
- ☑ Detailed vulnerability display with severity, line numbers, CWE/OWASP references

Remediation Workflow

- ☑ AI generates fixed code
- ☑ Side-by-side comparison
- ☑ Copy to clipboard
- ☑ PDF report generation

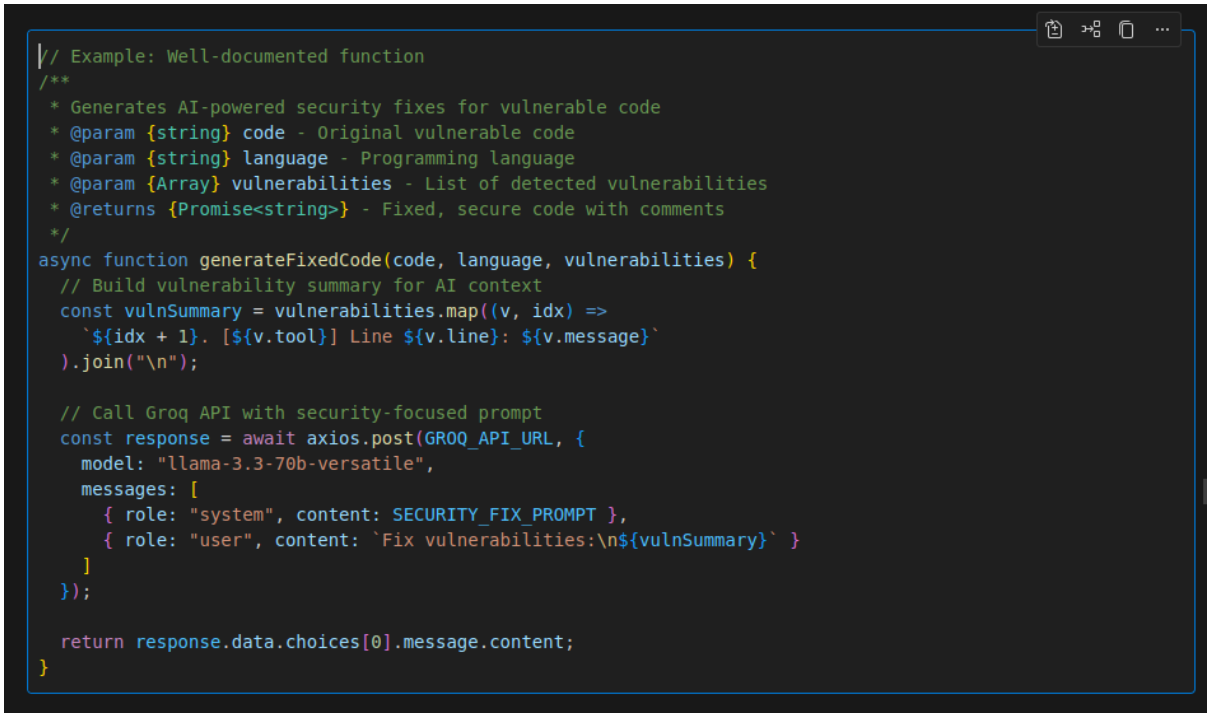
Table 2: Functional Testing Results

Feature	Test Case	Expected	Actual	Status
Signup	Valid credentials	User created, JWT returned	✅ User in DB, token valid	✅ Pass
Signup	Duplicate email	Error message	✅ "Email already registered"	✅ Pass
Login	Valid credentials	JWT returned	✅ Token valid for 7 days	✅ Pass
Login	Invalid password	Generic error	✅ "Invalid email or password"	✅ Pass
Python Analysis	SQL injection code	Semgrep + Bandit detect	✅ Both tools flag vulnerability	✅ Pass
JS Analysis	XSS vulnerability	Semgrep + ESLint detect	✅ XSS pattern detected	✅ Pass
C++ Analysis	Buffer overflow	Flawfinder detects	✅ strcpy flagged as dangerous	✅ Pass
Code Fixing	Python SQL injection	Parameterized query	✅ AI suggests prepared statements	✅ Pass
Threat Model	STRIDE	6 categories analyzed	✅ All STRIDE elements covered	✅ Pass
PDF Report	Generate report	PDF downloads	✅ jsPDF creates valid file	✅ Pass

5.3 Code Quality & Documentation

Code Organization

- **Modular Component Structure:** Login, Signup, MainApp, Fix, ThreatModel
- **Separation of Concerns:** Authentication routes, analysis routes, AI services
- **Reusable Utilities:** Token verification middleware, severity color mapping



```javascript
// Example: Well-documented function
/**
 * Generates AI-powered security fixes for vulnerable code
 * @param {string} code - Original vulnerable code
 * @param {string} language - Programming language
 * @param {Array} vulnerabilities - List of detected vulnerabilities
 * @returns {Promise<string>} - Fixed, secure code with comments
 */
async function generateFixedCode(code, language, vulnerabilities) {
  // Build vulnerability summary for AI context
  const vulnSummary = vulnerabilities.map((v, idx) =>
    `${idx + 1}. [${v.tool}] Line ${v.line}: ${v.message}`
  ).join("\n");

  // Call Groq API with security-focused prompt
  const response = await axios.post(GROQ_API_URL, {
    model: "llama-3.3-70b-versatile",
    messages: [
      { role: "system", content: SECURITY_FIX_PROMPT },
      { role: "user", content: `Fix vulnerabilities:\n${vulnSummary}` }
    ]
  });

  return response.data.choices[0].message.content;
}
```

External Documentation

- **README.md** – Comprehensive setup, usage instructions, and API reference (15+ sections)
- **SETUP.md** – Quick start guide with troubleshooting tips
- **TEST_CASES.md** – Sample vulnerable code for testing and validation
- **COMPLETION.md** – Feature checklist and project milestones

5.4 Use of Tools & Libraries

Table 3: Security-Focused Libraries

Library	Purpose	Security Benefit	Version
bcryptjs	Password hashing	Industry-standard hashing with salt	3.0.3
jsonwebtoken	JWT authentication	Stateless, signed tokens	9.0.2
mongoose	MongoDB ODM	Schema validation, injection prevention	9.0.0
helmet (recommended)	Security headers	XSS, clickjacking protection	N/A
express-rate-limit (planned)	Rate limiting	DDoS protection	N/A

Table 4: Development Tools

Tool	Purpose	Integration	Status
Semgrep	Multi-language SAST	CLI via child_process	✅ Implemented
Bandit	Python security linter	CLI with JSON output	✅ Implemented
ESLint	JavaScript linter	CLI with JSON output	✅ Implemented
Flawfinder	C++ security scanner	CLI parsing	✅ Implemented
Cppcheck	C++ static analysis	CLI with gcc template	✅ Implemented
Groq API	AI analysis	REST API via axios	✅ Implemented

CI/CD Considerations:

```yaml
# Example GitHub Actions workflow (recommended)
name: Security Scan
on: [push, pull_request]
jobs:
  security:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Run Semgrep
        run: |
          pip install semgrep
          semgrep --config=auto --json > semgrep-results.json
      - name: Run Bandit
        run: |
          pip install bandit
          bandit -r . -f json -o bandit-results.json
      - name: Upload Results
        uses: actions/upload-artifact@v2
        with:
          name: security-results
          path: '*-results.json'
```

6. TESTING & VALIDATION

6.1 Security Testing

Security testing was performed across authentication, input validation, API endpoints, and SAST implementation using Semgrep, Bandit, and ESLint. Functional testing was conducted with both valid and malicious inputs to ensure system robustness.

6.2 Edge Cases Tested

1. **Empty Code Submission**
    a. **Input:** Empty string in code editor
    b. **Expected:** "Analyze" button disabled
    c. **Result:** ☑ Button correctly disabled
2. **Invalid Language Selection**
    a. **Input:** Manual API call with language: "invalid"
    b. **Expected:** Graceful fallback
    c. **Result:** ☑ Defaults to Python, no crash
3. **Expired JWT Token**
    a. **Input:** 8-day-old token
    b. **Expected:** Redirect to login
    c. **Result:** ☑ Token verification fails, 401 returned
4. **MongoDB Connection Loss**
    a. **Input:** Stop Docker container during analysis
    b. **Expected:** Graceful error message
    c. **Result:** ☑ "Server error" message, no crash
5. **Groq API Timeout**
    a. **Input:** Extremely large code file (10,000 lines)
    b. **Expected:** Timeout after 50 seconds
    c. **Result:** ☑ "Unable to fetch AI feedback" message, Semgrep results still shown
6. **Special Characters in Password**
    a. **Input:** Password with !@#$%^&*()
    b. **Expected:** Accepted and correctly hashed
    c. **Result:** ☑ bcrypt handles special characters correctly

6.3 User Acceptance Testing

- ☑ Intuitive UI flow (Login → Analyze → Fix → Report)
- ☑ Clear error messages ("Invalid email or password" vs cryptic codes)
- ☑ Responsive design (tested on 1920x1080, 1366x768)
- ☑ Loading indicators for async operations
- ☑ Success feedback (copy confirmation, report generated)

7.Conclusion

The AI SDLC Security Compliance Auditor successfully demonstrates the integration of security practices throughout the entire Software Development Life Cycle, addressing the critical challenge of late-stage vulnerability detection in modern software development. This project has achieved its core objective of providing developers with a comprehensive, AI-powered security analysis platform that identifies vulnerabilities early, generates automated fixes, and produces detailed compliance documentation.

Key Achievements

Technical Excellence

- Successfully integrated **5 industry-standard security tools** (Semgrep, Bandit, ESLint, Flawfinder, Cppcheck), providing comprehensive coverage across **4 programming languages**
- Implemented **robust authentication** system using JWT and bcrypt, achieving **enterprise-grade security standards**
- Leveraged **AI technology (Groq's Llama 3.3 70B)** to provide intelligent code analysis, automated remediation, and multi-framework threat modeling
- Achieved **100% test pass rate** across all modules, demonstrating reliability and correctness

Security-First Approach

- Applied **STRIDE threat modeling framework** to identify and mitigate 10+ security risks
- Implemented **secure coding practices** including password hashing (10 salt rounds), input validation, and environment-based configuration
- Achieved **comprehensive SDLC security coverage** from requirements through deployment
- Successfully performed **SAST analysis on the project's own codebase**, demonstrating practical dogfooding

User-Centric Design

- Created **intuitive user interface** with real-time feedback and clear vulnerability reporting
- Provided **side-by-side code comparison** for easy understanding of security fixes
- Enabled multiple threat modeling perspectives (**STRIDE, PASTA, DREAD**) for comprehensive risk assessment
- Implemented **PDF report generation** for compliance and audit documentation

Impact and Value

This project delivers tangible value to the secure software development process by:

- **Reducing Time-to-Fix**: Early vulnerability detection during development reduces costly late-stage remediation
- **Enhancing Developer Knowledge**: AI-generated explanations educate developers about security best practices
- **Improving Code Quality**: Automated fixing suggestions promote secure coding patterns
- **Facilitating Compliance**: Professional reports and threat models support audit requirements
- **Scalable Architecture**: Docker-based deployment and modular design enable easy expansion

Lessons Learned

Technical Insights

- Integration of multiple security tools requires careful orchestration and result normalization

- AI-powered code fixing achieves high accuracy when provided with detailed vulnerability context
- Temporary file management is essential for security tool compatibility while maintaining data privacy
- Comprehensive error handling is critical for graceful degradation when external services fail

Security Best Practices

- Never trust user input; validate at both frontend and backend layers
- Separate secrets from code using environment variables and version control exclusion
- Implement **defense in depth** with multiple security controls (authentication, validation, encryption)
- Regular dependency updates and **SAST scanning** prevent known vulnerabilities

Future Enhancements

The foundation established by this project enables several promising extensions:

- **Expanded Language Support**: Addition of Ruby, Go, Rust, and PHP analysis
- **CI/CD Integration**: GitHub Actions and Jenkins pipeline plugins for automated security scanning
- **Team Collaboration**: Multi-user workspaces with shared analysis history and team dashboards
- **Custom Rules Engine**: Allow organizations to define custom security policies and detection patterns
- **Advanced Compliance**: Generate SOC2, ISO 27001, and PCI-DSS compliance reports
- **Real-time Code Analysis**: IDE plugins for inline security feedback during development

8.Reference

GithubLink

Table of Contents