# CodeCove : A Code Review Tool

Aayush Parmar*, Birudugadda Srivibhav†, Bhoumik Patidar‡, Dewansh Kumar§
*22110181, †22110050, ‡22110049, §22110071
*Indian Institute of Technology Gandhinagar*
Email: {aayush.parmar,birudugadda.srivibhav,bhoumik.patidar,dewansh.kumar}@iitgn.ac.in

***Abstract*—This report documents the development and progress of our Code Review Tool, a web application designed to facilitate collaborative code review workflows in distributed teams. The tool integrates Git repository management, pull request workflows, inline commenting, SSH key management, and static analysis, combining both backend and frontend components. The backend is hosted on AWS EC2 and the frontend is built in React. This report outlines project milestones, architecture, features implemented, networking aspects, server-side implementation, and workflow.**

***Index Terms*—Code Review, Git, SSH Authentication, Client-Server Architecture, RESTful API, React, Node.js, Static Analysis, Pull Requests, Diff Visualization**

## 1 Introduction

The Code Review Tool is a web application designed to facilitate collaborative code review workflows in a distributed team environment. Inspired by tools such as Gerrit, the tool integrates Git repository management with pull request (PR) workflows, code submission, and SSH key management. The application consists of both backend and frontend components, with the backend hosted on an AWS EC2 instance and the frontend developed using React.

## 2 Project Milestones

### *2.1 Targets Achieved*

- **Repository Management**: Automated creation/listing of bare repositories with conflict prevention and naming conventions.
- **Pull Request Workflow**: Full PR lifecycle implementation including conflict detection and atomic merges.
- **Gitolite Permission System**: Granular access control with dynamic permission updates and branch/repository-level policies.
- **Static Analysis Pipeline**: Integrated Bandit, Flake8, and Cppcheck for automated code quality checks.
- **Git Integration**: Endpoints for commit history retrieval, diff generation, and branch comparisons.

## 3 Architecture Overview

### *3.1 Backend*

- Node.js with Express for REST API endpoints.
- Sequelize ORM for MySQL database interactions.
- NodeGit for Git operations (cloning repositories, merging branches, fetching diffs).
- JWT for authentication.
- Hosted on AWS EC2 (Ubuntu) with bare repositories in `/home/git/repositories`.
- CORS and security group settings ensure remote access.

### *3.2 Frontend*

- React with React Router for client-side routing.
- Axios for API calls (configured in api.js).
- Developed locally, communicates with backend via AWS public IP.
- Includes pages for login/registration, dashboard, repository management, pull request management, and detailed PR view.

## 4 Features Implemented

### *4.1 Repository Management*

The Code Review Tool provides an intuitive interface for repository management, enabling teams to create and explore Git repositories without manual server-side operations.

**Functionality:**

- Teams can create new repositories via a web form, specifying the repository name.
- The system ensures the repository name is unique and appends the `.git` extension if not present.
- Bare repositories are initialized using NodeGit in `/home/git/repositories` .
- Existing repositories are listed with metadata such as creation date and owner.

**Code Snippet:**

```
async function createRepo(req, res) {
  try {
    const { name } = req.body;
    if (!name) {
      return res.status(400).json({ error: "Repository name is required" });
    }
    const repoName = name.endsWith('.git') ? name : `${name}.git`;
    const newRepoPath = path.join(REPO_BASE_PATH, repoName);
    try {
      await fs.access(newRepoPath);
      return res.status(400).json({ error: "Repository already exists" });
    } catch (err) {
      // Repository doesn't exist; continue to create.
    }
    const repo = await NodeGit.Repository.init(newRepoPath, 1);
    res.json({ message: "Repository created successfully", repository: repoName });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
}
```

**Frontend:**

- The `RepositoryManagement.jsx` component provides a user-friendly interface for creating and viewing repositories.
- Users can browse, filter, and select repositories for further actions.

**Backend Implementation:**

- Endpoint: `POST /api/repos/create`

- Validates repository name, checks for duplicates, and initializes the repository.
- Updates the repository list for all users with appropriate permissions.

### *4.2 Pull Request (PR) Workflow*

The tool implements a robust pull request workflow, central to collaborative code review and integration.

**Functionality:**

- Developers create PRs to propose merging changes from a source branch to a target branch.
- Each PR includes a title, description, and links to commit history and diffs.
- Reviewers can approve, request changes, or merge PRs based on their permissions.
- PR status is tracked (open, approved, merged, closed).

**Backend Implementation:**

- The `PullRequest` model records repository, source/target branches, author, status, and metadata.
- Endpoints:
  - `POST /api/prs/create`: Create a new PR.
  - `GET /api/prs`: List all PRs with filtering options.
  - `POST /api/prs/:id/approve`: Approve a PR.
  - `POST /api/prs/:id/merge`: Merge a PR using NodeGit in a temporary working clone.

**Merge Endpoint (Simplified):**

```
exports.mergePR = async (req, res) => {
  // Clone the bare repository into a temporary directory,
  // checkout target branch, merge source branch, push merged changes,
  // update PR status to "merged".
};
```

**Frontend:**

- `PRDashboard.jsx`: Lists PRs, shows status, and provides actions for approval/merge.
- `PRDetail.jsx`: Presents PR details, commit history, code diffs, and review actions.

### *4.3 Static Analysis for Code Quality and Security*

To enforce code quality and security, the system integrates an automated static analysis pipeline.

**Workflow:**

- When a PR is created or updated, static analysis can be triggered from the PR detail view.
- The backend clones the repository and identifies relevant files (Python, C/C++).
- Tools used:
  - **Bandit**: Scans Python code for security vulnerabilities.
  - **Flake8**: Checks Python code for style and quality issues.
  - **Cppcheck**: Analyzes C/C++ code for logical and security errors.
- Results are aggregated into a structured report and displayed on the frontend.
- Temporary files and working directories are cleaned up after analysis.

**Frontend:**

- "Run Static Analysis" button in the PR detail page.
- Results are presented in a tabular format, highlighting severity, file, and line number.

### *4.4 Git Integration Endpoints*

The backend exposes endpoints for advanced Git operations, enabling rich code review features.

**Functionality:**

- **Commit History:**
  `GET /api/repos/:repoName/commits`
  - Retrieves the full commit history of a repository.
  - Handles cases where HEAD is missing by falling back to the main branch.
- **Diff Information:**
  `GET /api/repos/:repoName/diff/:commitSha`
  - Generates and returns file diffs between a commit and its parent.
  - Supports side-by-side and inline diff visualization.

**Code Snippet (Commit History with Fallback):**

```
async function getCommits(req, res) {
  const repo = await NodeGit.Repository.open(repoPath);
  let headCommit;
  try {
    headCommit = await repo.getHeadCommit();
  } catch (e) {
    headCommit = await repo.getBranchCommit('main');
  }
  // Process commit history...
}
```

**Frontend:**

- Commit history and code diffs are visualized with color-coded highlights for additions and deletions.
- Reviewers can navigate between commits and view detailed changes.

### *4.5 SSH Key Management*

Secure repository access is enforced via SSH key authentication.

**Functionality:**

- Users submit their SSH public key during registration or update it via a dedicated interface.
- The backend validates the key format and updates the user's record.
- The authorized_keys file on the server is synchronized to enable access.
- Each SSH key is linked to a user, supporting secure and auditable access.

**Frontend:**

- `SshKeyUpdate.jsx`: Provides a form for users to paste or update their SSH public key.
- Users receive confirmation and can view their current key.

### *4.6 Permission Control System with Gitolite Integration*

To provide granular, secure access control, the tool integrates with **Gitolite**, a powerful Git server access management system.

**Overview:**

- Gitolite allows administrators to define fine-grained permissions (read, write, force-push) on repositories and branches.
- Permissions are managed via SSH key hashes, mapped to users and groups.
- The system supports per-repository, per-branch, and per-user permission assignment.

**Permission Levels:**

- **R (Read-only):** Users can only clone and pull from the repository.
- **RW (Read-Write):** Users can push changes to the repository.
- **RW+ (Read-Write Plus):** Users can force-push and delete branches.

**Implementation:**

- Users submit their SSH key via the frontend; a hash of the key is generated.
- Administrators assign permission levels (R, RW, RW+) using the Permissions interface.
- The backend updates the `gitolite.conf` file, mapping each key hash to the assigned permission.
- Example configuration:

```
repo gitolite-admin
    RW+ = admin
repo testing
    RW+ = hash1
    RW+ = hash2
repo test1
    R = hash1
repo debug
    RW+ = hash2
    R = hash3
```

- Gitolite enforces these permissions automatically during all Git operations via SSH.

**Frontend:**

- The `Permissions.jsx` component allows admins to visually assign and update permissions for each user and repository.
- Permission changes are reflected in real-time and confirmed to the admin.

**Security and Audit:**

- All permission changes are logged for audit purposes.
- Default policies deny unauthorized access, ensuring repository integrity.
- The system supports group-based permissions for efficient management of large teams.

**Summary:**
These features collectively provide a secure, collaborative, and efficient environment for distributed software development, combining modern code review practices with robust access control and automation.

## 5 Networking Aspects

### *5.1 Network Configuration and Connectivity*

- Cloud-Hosted Backend: The backend, including the Git server, is hosted on an AWS EC2 instance. It is accessible via a public IP address on a designated port.
- Frontend Connectivity: The frontend is developed using React and runs locally. It is configured to communicate with the cloud-hosted backend.
- CORS Policy: CORS is enabled in the backend using the cors middleware.
- Security Groups: AWS EC2 security groups are configured to allow incoming traffic on the necessary ports (e.g., port 5000 for API calls and port 22 for SSH).
- SSH Key Management: Users can update their SSH public keys through the web interface; keys are synchronized with the authorized_keys file.

### *5.2 Networking and Security Considerations*

- Backend Accessibility: Hosted on an AWS EC2 instance with a public IP, ensuring remote accessibility via a designated port.
- CORS Configuration: Configured via Express middleware to allow API requests from local frontend environments and specified origins.
- AWS Security Groups: Inbound rules are set to allow traffic on port 5000 (for API) and port 22 (for SSH), with restrictions based on IP addresses.
- SSH Key Automation: Users update their SSH keys through the web interface; keys are automatically appended to the Git user's authorized_keys file.
- Network Reliability: The system ensures consistent connectivity between the local frontend and cloud backend.

## 6 Server-Side Implementation Details

### *6.1 Git Server Integration*

- Repositories are created as bare repositories in the directory `/home/git/repositories`.
- NodeGit is used for:
  - Cloning repositories (using a temporary working clone for operations that require a working directory).
  - Retrieving commit histories and file diffs.
  - Merging feature branches into the main branch during pull request (PR) workflows.

- Merge operation: Clone the bare repository into a temporary directory, check out the target branch, merge the source branch, push the changes back, and update the PR status in the database.

**Example Code Snippet (Creating a Bare Repository):**

```
const NodeGit = require("nodegit");
const path = require("path");
const fs = require("fs-extra");

const REPO_BASE_PATH = "/var/lib/git";

async function createRepo(name) {
  const repoName = name.endsWith('.git') ? name : `${name}.git`;
  const newRepoPath = path.join(REPO_BASE_PATH, repoName);
  await fs.access(newRepoPath).catch(() => {}); // Ensure repo doesn't exist
  const repo = await NodeGit.Repository.init(newRepoPath, 1);
  return repo;
}
```

### *6.2 Database Integration*

- MySQL database is used to store application data, including user information, code submissions, pull requests, and comments.
- Sequelize ORM is used for model definition and data operations.
- JWT-based authentication secures API endpoints.
- Database synchronization updates the schema to reflect model changes.

**Example Code Snippet (Database Configuration):**

```
const { Sequelize } = require('sequelize');
require('dotenv').config();

const sequelize = new Sequelize({
  host: process.env.MYSQL_HOST,
  username: process.env.MYSQL_USER,
  password: process.env.MYSQL_PASSWORD,
  database: process.env.MYSQL_DB,
  port: process.env.MYSQL_PORT,
  dialect: 'mysql',
  logging: true,
});

module.exports = sequelize;
```

### *6.3 Interconnection of Components*

- Express API routes handle repository, PR, and SSH key management.
- SSH keys updated via API and synced to authorized_keys.
- Git server and database interact seamlessly with the backend API.

### *6.4 Permission Control System with Gitolite Integration*

*6.4.1 Gitolite Architecture:* The tool integrates Gitolite for granular access control, leveraging its SSH-based authentication and flexible permission management. Key components include:

- **SSH Key Mapping:** User public keys are hashed using SHA-256 and mapped to permissions in the Gitolite configuration.
- **Permission Levels:**
  - **R (Read-only):** Clone/fetch access only
  - **RW (Read-Write):** Push access with fast-forward restrictions
  - **RW+:** Force push and branch deletion capabilities
- **Configuration Management:** Dynamic updates to `gitolite.conf` through automated backend processes.

*6.4.2 Implementation Workflow:*

1) **SSH Key Submission:** Users submit public keys via the `SshKeyUpdate` interface.
2) **Hash Generation:** Backend generates SHA-256 hash of the public key.
3) **Config Update:** Hashes and permissions are written to Gitolite's configuration:

```
repo test-repo
    RW+ = 1a72d34...1db53
    R   = ea1b7cc...53a
```

4) **Access Enforcement:** Gitolite validates SSH connections against stored hashes during Git operations.

The system maintains a strict mapping between SSH key hashes and repository permissions, ensuring secure access control while preserving Git's native workflow efficiency. This integration enables enterprise-grade permission management while remaining transparent to end users during regular Git operations.

## 7 Workflow of the Code Review Tool

The Code Review Tool implements a structured, collaborative workflow that guides teams from repository creation to code integration. This section details each stage of the process, illustrating the journey of code from inception to deployment, and highlights the role of automation, permission control, and review in maintaining code quality.

### *7.1 Step 1: Repository Creation*

The workflow begins with repository creation, typically performed by a team lead or project administrator. Using the web interface, the lead specifies the repository name and initializes it on the server as a bare Git repository. This step establishes the central location for all subsequent code collaboration.

- The user navigates to the "Repository Management" section of the application.
- A form is provided to input the repository name, ensuring naming conventions and uniqueness.
- Upon submission, the backend uses NodeGit to create a bare repository in the server's repository directory (e.g., `/home/git/repositories`).
- The new repository is immediately listed among existing repositories, with metadata such as creation date and owner.

**Significance:** Establishing a centralized repository ensures all project members work from a single source of truth, simplifies access control, and enables structured collaboration from the outset.

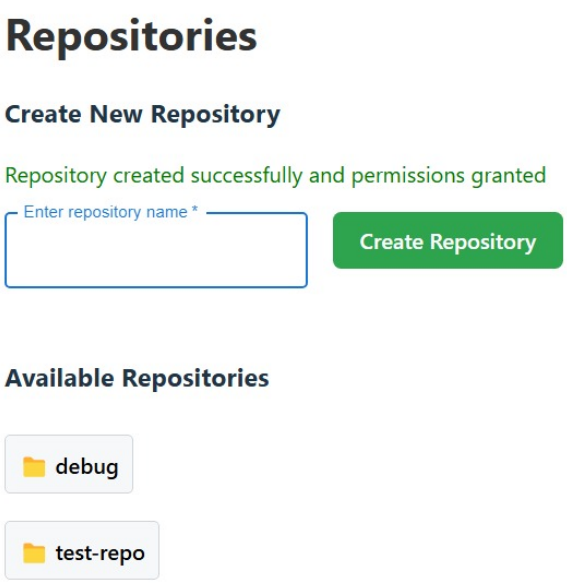


Fig. 1. Repository Creation Screen: Web interface for initializing a new test-repo.

### *7.2 Step 2: Code Submission and Branching*

Once a repository is created, developers can begin contributing code. This process involves cloning the repository, creating feature branches, and pushing commits.

- Developers obtain the SSH clone URL for the repository from the web interface.
- Using their registered SSH keys, developers execute the clone command:

```
git clone git@<ip>:<repo>.git
```

- Gitolite enforces permission checks, ensuring only authorized users can push to specific branches.

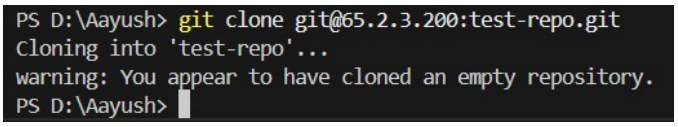


Fig. 2. Cloning and Branching: Developer workflow for code submission.

**Significance:** Branch-based development enables parallel workstreams, isolates changes, and facilitates organized integration via pull requests.

### *7.3 Step 3: Pull Request Creation*

After completing work on a feature branch, a developer initiates a pull request (PR) through the web interface to propose merging their changes into a target branch (commonly `main` or `develop`).

- The developer accesses the ”Pull Requests” section and selects ”Create Pull Request.”
- The form prompts for:
  - Source branch (the feature branch to merge from)
  - Target branch (the branch to merge into)
  - PR title and detailed description
- The PR is submitted and enters the review queue, visible to all reviewers and team leads.
- The PR dashboard displays status, author, reviewers, and links to commit history and diffs.

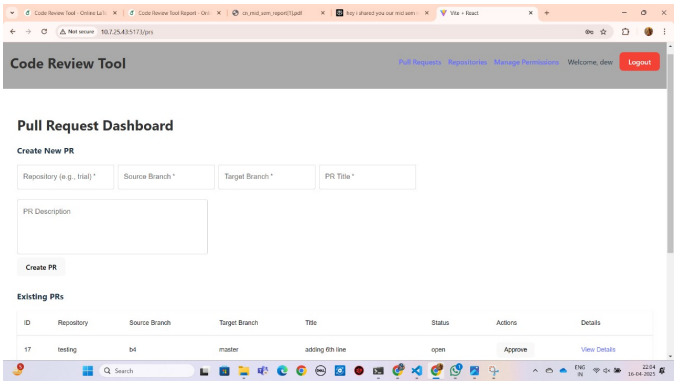


Fig. 3. Pull Request Creation: Submitting a new PR for review.

**Significance:** Pull requests formalize the review process, provide a forum for discussion, and ensure traceable, auditable code integration.

### *7.4 Step 4: Code Review and Discussion*

Reviewers, typically senior developers or designated peers, examine the proposed changes for correctness, style, and adherence to project standards.

- Reviewers access the PR detail view, which presents:
  - A summary of changes (commit list, affected files)
  - Side-by-side or inline code diffs with color-coded highlights for additions and deletions
  - Inline and general commenting features for detailed feedback
- Reviewers may request changes, approve, or reject the PR.
- The discussion thread captures all feedback, questions, and resolutions, creating a persistent review history.

**Significance:** This step enforces code quality, encourages knowledge sharing, and reduces the likelihood of introducing defects into the main codebase.

### *7.5 Step 5: Static Analysis Integration*

Before a PR can be approved, the tool integrates automated static analysis to detect security vulnerabilities, code style violations, and maintainability issues.

- The reviewer or author triggers static analysis via a dedicated button in the PR interface.

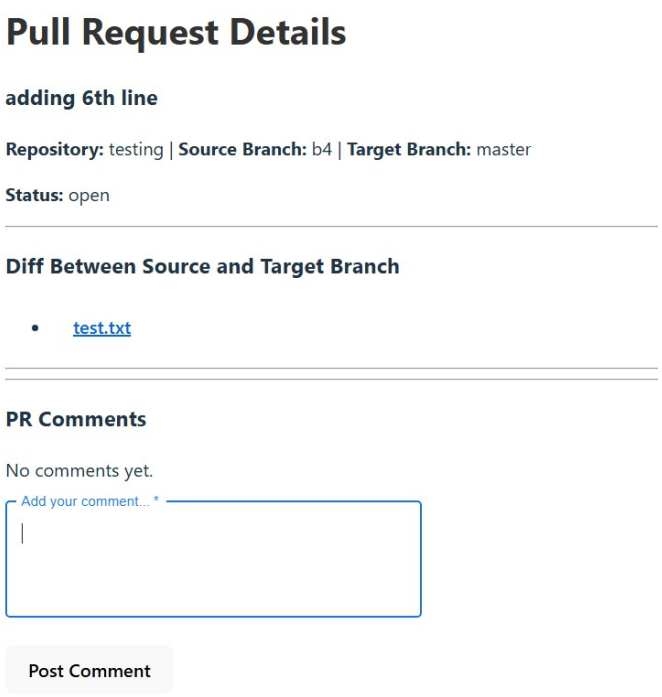


Fig. 4. Code Review Interface: Reviewing changes and providing feedback.

- The backend executes tools such as Bandit (Python security), Flake8 (Python style), and Cppcheck (C/C++ analysis) on the PR's code.
- Results are parsed, summarized, and displayed in the PR view, highlighting issues by severity and location.
- Developers are prompted to resolve flagged issues before proceeding.

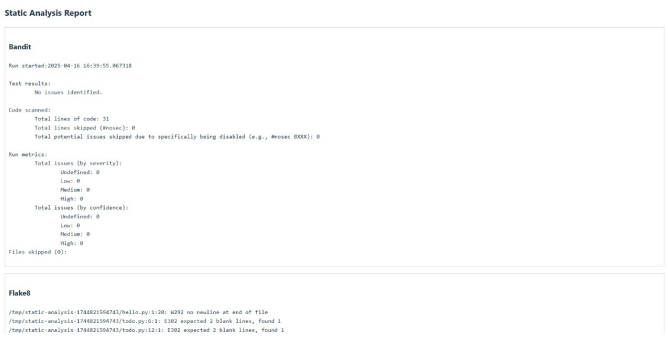


Fig. 5. Static Analysis Results: Automated code quality and security checks.

**Significance:** Automated analysis reduces manual review effort, ensures compliance with best practices, and catches subtle defects early in the workflow.

### *7.6 Step 6: Approval and Merge*

Once the PR passes both human review and automated checks, it can be approved and merged into the target branch.

- Reviewers or team leads approve the PR through the interface.
- The backend creates a temporary working clone, merges the feature branch into the target, resolves conflicts if any, and pushes the result.
- The PR status is updated to "merged," and all participants are notified.
- The merged code becomes part of the main codebase, available for further development or deployment.

**Significance:** This step ensures only reviewed and validated code enters the main branch, maintaining project stability and traceability.

### *7.7 Step 7: Permission and Access Control*

Throughout the workflow, Gitolite enforces repository and branch-level permissions:

- User SSH keys are mapped to specific permission levels (R, RW, RW+) in the Gitolite configuration.
- The Permissions interface allows administrators to assign or revoke access for users or groups.
- All Git operations (clone, push, merge) are authenticated and authorized via SSH and Gitolite rules.
- Unauthorized actions are automatically blocked, and audit logs are maintained for compliance.

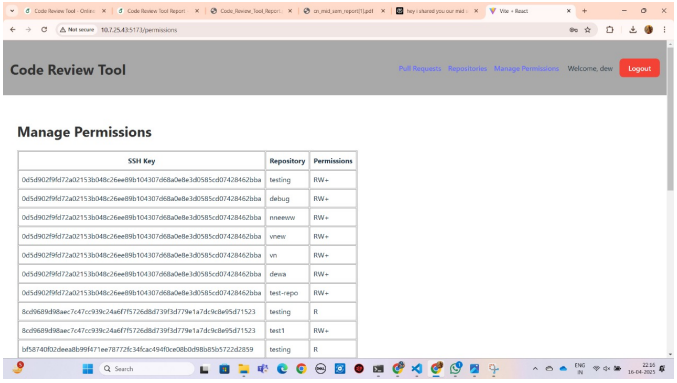


Fig. 6. Permission Management: Granular access control via Gitolite integration.

**Significance:** Granular permission control ensures security, prevents unauthorized changes, and supports collaborative workflows across teams and projects.

**Summary:** The Code Review Tool's workflow provides a robust, repeatable process for collaborative software development. By combining clear role separation, automation, permission control, and comprehensive review features, it ensures high-quality, secure, and maintainable code integration for distributed teams.

## 8 Lessons Learned

The development of this Code Review Tool provided numerous valuable insights and learning opportunities for our team. We encountered and overcame several technical challenges,

refined our understanding of modern web development practices, and gained practical experience with Git integration in web applications.

### A. Technical Insights

- **Git Integration Complexity**: Integrating Git operations into a web application proved more complex than anticipated. NodeGit provided powerful capabilities but required careful error handling and asynchronous operation management. We learned to handle edge cases such as empty repositories, missing branches, and concurrent operations.
- **Authentication Flow Design**: Implementing a secure authentication system with JWT tokens and SSH key integration required careful coordination between frontend and backend components. The AuthContext provider pattern combined with protected routes provided an elegant solution for maintaining authenticated state across the application.
- **React Route Protection**: Creating an effective route protection system using React Router demanded thoughtful state management and loading indicators. Our ProtectedRoute component implementation successfully handles authentication verification, loading states, and redirects.
- **Diff Visualization**: Developing an intuitive diff visualization system required creative approaches to transform raw Git diff output into user-friendly interfaces. We gained experience with processing and formatting complex textual data for visual presentation.

### B. Architectural Decisions

- **Component Structure**: Organizing our frontend into pages, components, and context providers improved code maintainability. We learned the value of clear separation between UI elements, data fetching, and business logic.
- **API Design**: Designing a comprehensive REST API for Git operations required careful consideration of resource naming, parameter handling, and error responses. We gained appreciation for consistent API patterns and thorough documentation.
- **State Management**: Using React Context for global state management (particularly authentication state) proved more efficient than prop drilling for our application structure. This approach reduced complexity in deeply nested component hierarchies.
- **Permission System**: Implementing a granular permission system using Gitolite demonstrated the importance of mapping application-level permissions to underlying system capabilities. The permission management interface required careful coordination between frontend controls and backend enforcement.

### C. Development Workflow

- **Client-Server Debugging**: Debugging interactions between frontend and backend components required systematic logging and error tracking. The navigation and authentication state logging in App.jsx proved invaluable for troubleshooting route protection issues.
- **Testing Git Operations**: Testing Git-related functionality demanded careful setup of test repositories and consideration of various edge cases. We learned to create comprehensive test scenarios for repository operations.
- **Security Considerations**: Implementing secure authentication and authorization required vigilance across all application layers. We gained experience with JWT token management, SSH key validation, and route protection strategies.
- **Error Handling**: Developing robust error handling for Git operations and API requests improved application reliability. We learned to provide informative error messages while protecting sensitive system details.

Our experience developing this Code Review Tool has enhanced our understanding of full-stack web application development, particularly in the areas of authentication systems, Git integration, and collaborative workflows. These lessons will inform our approach to future software projects, especially those involving version control systems and team collaboration tools.

## 9 Individual Contributions

Each team member contributed to different components of the project:

- **Aayush Parmar**
  - Implemented SSH key management functionality
  - Created the user interface for repository exploration
  - Built the file viewer with syntax highlighting
- **Birudugadda Srivibhav**
  - Designed and implemented the pull request workflow
  - Created the code comparison interface
  - Implemented the static code analysis integration
  - Built the PR comment system
- **Bhoumik Patidar**
  - Developed the repository management backend
  - Implemented Gitolite integration for permissions
  - Created the permission management interface
  - Built the merge conflict detection and resolution system
- **Dewansh Kumar**
  - Designed the overall system architecture
  - Created the RESTful API infrastructure
  - Implemented the commit history tracking system
  - Developed the branch management functionality

## 10 Conclusion

In this project, we developed a comprehensive Code Review Tool that integrates repository management, a complete pull request workflow, and robust Git operations with secure SSH key management. The system leverages modern technologies including Node.js, Express, Sequelize, and NodeGit on the backend, and React with React Router on the frontend.

Key features include:

- Automated creation and listing of bare repositories.
- An end-to-end pull request workflow for code submission, review, approval, and merging.
- Integration of static analysis tools for ensuring code quality and security.
- Automated SSH key management for secure Git operations.

The tool has been designed with scalability in mind, supporting multi-team collaboration and laying a foundation for future enhancements such as inline diff commenting, advanced merge conflict resolution, role-based access control, and real-time notifications.

Overall, the Code Review Tool streamlines the code review process while enforcing industry best practices, making it a valuable asset for modern software development teams.

## REFERENCES


[1] React Documentation. https://react.dev/
[2] Node.js Documentation. https://nodejs.org/en/docs
[3] Express.js Documentation. https://expressjs.com/en/starter/installing.html
[4] JWT.io. JSON Web Tokens - Introduction. : https://jwt.io/introduction
[5] NodeGit Documentation. https://www.nodegit.org/api/
[6] Gitolite Documentation. https://github.com/sitaramc/gitolite
[7] Git Documentation. https://git-scm.com/doc