CollabCode: A Real-Time Collaborative Coding Platform with Artificial Intelligence Code Assistance and Secure Multi-Language Execution

Yashawantha H G[#1], Sakeeb Pasha[#2], Nivedita V S[#3]

[#]Department of Information Science and Engineering, CMR Institute of Technology, Bengaluru, India

[1]yahg23ise@cmrit.ac.in, [2]sakeeb.ise.24le@cmrit.ac.in, [3]nivedita@cmrit.ac.in

Abstract

This work presents CollabCode, a browser-based platform for collaborative programming. It combines shared code editing, artificial intelligence assisted coding, server-side execution, persistent sessions, and a Secure Interview Mode in one environment. The system is implemented using React.js, Node.js, Socket.IO, MongoDB, Monaco Editor, and Docker. Editor changes are transmitted through persistent WebSocket connections, while AI requests use a separate communication path so coding activity can continue without interruption. Programs written in C, C++, Java, and Python are executed in isolated containers with resource and network restrictions. Users can create coding rooms and recover stored sessions. In the conducted tests, synchronization remained below 50 ms for rooms with up to ten concurrent users, execution responses remained below 1.5 seconds, and message delivery reliability stayed above 98%. Secure Interview Mode records browser and session events and provides the interviewer with a security view. The platform is intended for collaborative programming, pair programming, and technical interviews.

Keywords—CollabCode; Real-Time Collaboration; WebSocket; Socket.IO; Monaco Editor; Docker; Artificial Intelligence Code Assistance; Collaborative Programming; React.js; Node.js; MongoDB; Code Execution; Secure Coding; Online Technical Interview

I. INTRODUCTION

Cloud services and browser-based development have changed the way programmers work together. Instead of depending on a single physical development machine, users can access programming tools through a browser and participate in the same coding session from different locations. This approach is useful in remote software development, pair programming, classroom activities, project demonstrations, and technical interviews [1]. In such situations, collaboration depends on how quickly changes made by one participant become visible to the others. Even a small delay can make the editing process less natural, particularly when two or more users are working on the same section of a program.

Traditional development workflows are generally organized around local editing and version-control systems. Git and similar tools are highly useful for maintaining project history, creating branches, reviewing changes, and merging completed work. However, their usual commit, push, and merge cycle does not provide continuous synchronization of every keystroke between users. For a live coding session, participants may therefore need additional communication or coordination to understand what another person is currently changing [2]. Existing browser-based collaborative editors address part of this problem, but some solutions depend on specific environments or include a broad set of features that may not be necessary for smaller teams, student projects, or short technical assessments [3].

Another requirement is the ability to execute code without moving the user to a separate development environment. A collaborative editor becomes more useful when participants can write, modify, and test programs within the same session.

At the same time, executing source code submitted by users introduces security and resource-management concerns. The execution service must prevent submitted programs from directly affecting the main application and should control execution time, resource consumption, and network access. Container-based isolation provides a practical approach for separating submitted programs from the application infrastructure [4].

Artificial intelligence (AI)-assisted programming introduces another layer to the collaborative workflow. Modern coding tools can provide context-aware completions, explanations, and review-oriented suggestions [5]. However, AI requests can require additional processing time and should not interfere with the continuous exchange of editor events. A collaborative platform therefore needs to treat AI assistance as a supporting service rather than allowing it to become part of the critical synchronization path. Separating these communication flows helps maintain the responsiveness of the shared editor while suggestions are being generated.

CollabCode was developed to combine these requirements in a single browser-based environment. The platform uses React.js for the user interface and Node.js with Express.js for server-side services. Socket.IO provides persistent communication for real-time collaboration, while MongoDB stores room and session information. Monaco Editor supplies the programming interface, and Docker is used to execute submitted programs in isolated environments. The system also includes AI-assisted suggestions and a dedicated Secure Interview Mode for controlled coding assessments [6]. A

participant can enter a room using a generated identifier, edit the shared program, communicate with other participants, execute supported languages, and recover stored session information when required.

The design also separates normal collaborative activity from interview-specific monitoring. In an ordinary coding room, users can work together without additional browser restrictions. When Secure Interview Mode is enabled by an interviewer, selected browser and session events can be recorded and displayed through a security dashboard. These events include changes in tab visibility, window focus, fullscreen state, clipboard activity, and other configured interactions. The purpose of this mechanism is to provide the interviewer with additional session information for review rather than automatically treating an individual browser event as proof of misconduct.

The main components implemented in CollabCode are:

- A two-stream communication design that keeps frequent editor updates separate from AI requests.
- A browser editor based on Monaco Editor with Socket.IO support for shared editing and cursor presence.
- Container-based execution for C, C++, Java, and Python with resource and network restrictions.
- MongoDB-backed rooms that retain session information and code snapshots for recovery.
- Inline suggestions generated from the current coding context [5].
- A Secure Interview Mode that records selected browser events, manages warnings, protects sessions, and gives the interviewer a live security view.

II. Literature Review

Earlier work has examined different ways of supporting shared programming over a network. Virdi et al. [1] described CodeAndCollab as a web-based multi-user coding system with secure execution. Their work shows that synchronous editing is practical, while also pointing to difficulties that appear as the number of simultaneous users grows. Xu et al. [7] explored an event-driven distributed programming framework and considered conflict resolution as part of maintaining responsive collaboration.

The way simultaneous edits are reconciled is another central issue. Sun and Sun [2] studied Operational Transformation (OT), which adjusts concurrent changes so that group editing can continue. Shapiro et al. [8] introduced Conflict-Free Replicated Data Types (CRDTs) as another approach, with eventual consistency as a key property. Zhang et al. [9] examined intelligent conflict-resolution methods for programming environments and reported that adaptive techniques can reduce merge problems when many users edit concurrently.

For the communication layer, persistent WebSocket connections are well suited to applications that need frequent event updates. Pimentel and Nickerson [6] compared WebSockets with HTTP polling for real-time data exchange and reported lower latency with WebSockets. Fan et al. [10] developed CoVSCode for lightweight collaborative programming and also investigated shared locking for semantic conflict prevention [11]. Their results indicate that excessive locking can itself become a bottleneck. Nakamura et al. [12] investigated semantic methods for reducing merge conflicts in collaborative IDEs.

The editor is another important part of the user experience. Monaco Editor [13], the editor technology used by Visual Studio Code, provides browser-based syntax highlighting, IntelliSense, and support for multiple programming languages. Wang et al. [14] studied editing conflicts in collaborative computational notebooks and found that features such as cursor awareness and conflict indicators help users understand simultaneous changes.

Running submitted programs introduces a separate security concern because the code comes from users. Bernstein [4] discussed containerization with Docker as a way to isolate workloads. Park et al. [15] examined deployment approaches for collaborative programming systems and considered cloud execution as a means of improving access to such tools.

AI-assisted programming has also become part of modern development environments. Chen et al. [5] evaluated language models trained on code and reported benefits from context-aware completion. Rasheed et al. [16] examined large language model (LLM)-based code review and its ability to identify bugs and code smells. Kawalerowicz et al. [17] reviewed the use of LLMs as code-review agents, while AICodeReview [18] provided an example of an AI-supported review workflow.

Other work is relevant to the infrastructure used by CollabCode. Kulikov et al. [19] investigated frameworks for distributed development with an emphasis on robustness. Burns et al. [20] discussed large-scale deployment using Kubernetes and related cloud-native mechanisms. Chodorow [21] describes MongoDB as a practical NoSQL database for application data such as sessions, snapshots, and histories, which matches several of the persistence requirements of a collaborative editor.

III. Proposed System Architecture of CollabCode

The implemented system uses a client-server arrangement in which each major function has a separate responsibility. React.js provides the frontend, while Node.js and Express.js handle server-side requests. Socket.IO carries live collaboration events, MongoDB stores persistent data, and Docker provides the isolated environment used to run submitted programs.

A. System Overview

A user can enter a coding room and work on the shared editor with the other participants. The implementation keeps editor events and AI requests on separate communication paths. This separation is intended to prevent a burst of editing activity from delaying an AI response, or an AI request from interrupting the normal synchronization path. The complete arrangement is shown in Fig. 1.

ISSN: 3107-6513

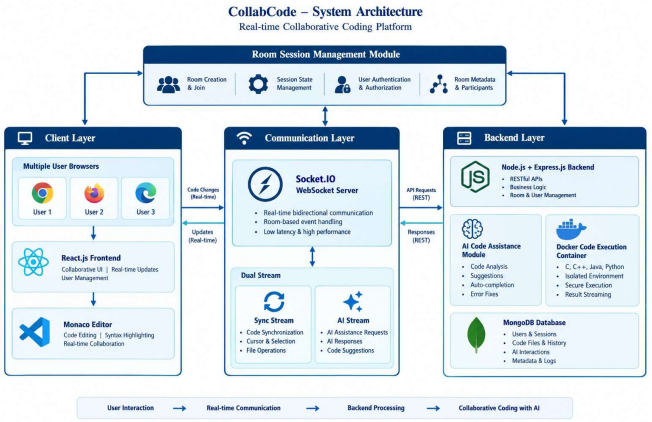

Fig. 1. System Architecture of CollabCode.

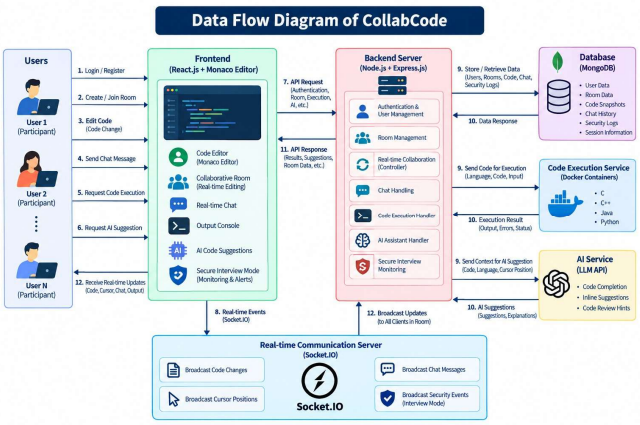

Fig. 2. Data Flow Diagram of CollabCode.

B. *Real-Time Synchronization Module*

Socket.IO maintains the live connection between the participants in a room. An edit made in Monaco Editor is converted into a change event and sent to the other connected clients, along with cursor and related editor events [6]. When someone enters an already active room, the server obtains the latest stored code state from MongoDB and sends it to that participant. This gives a newly joined user the same starting state as the existing members [21].

C. *Code Execution Module*

Program execution takes place on the server rather than directly inside the browser. The submitted source is sent to a language-specific Docker environment for C, C++, Java, or Python. Resource limits, execution timeouts, and network restrictions are applied to the container [4]. After execution, the service sends the program output or error information back to the frontend without blocking the rest of the collaborative session.

D. *AI Code Assistance Module*

The AI component uses a large language model to generate completion suggestions from the code being edited [5]. AI requests are sent separately from the live synchronization traffic, so generating a suggestion does not have to stop the exchange of editor events. Returned suggestions are shown as inline hints through the Monaco Editor interface [13]. The implementation also includes basic review-oriented hints based on approaches reported for automated code review [16], [17].

E. *Session Management and Data Persistence*

MongoDB is used for information that needs to survive beyond an individual connection, including room details, session information, chat history, and saved code states [21]. Each room receives a generated identifier that is used when participants access the session. Because code states are stored, a disconnected participant can recover the previous session state after reconnecting.

F. *Deployment Architecture*

For deployment, the application is packaged with Docker and can be managed through Kubernetes when a cloud deployment requires scaling [20]. The movement of data between the main modules is presented in Fig. 2.

IV. Secure Interview and Coding Mode

CollabCode includes a separate Secure Interview Mode for situations in which the coding room is used as an assessment environment. It is enabled by the interviewer, so ordinary collaborative rooms do not need to use the additional restrictions. Once enabled, the mode combines browser event collection with server-side checks and a dashboard through which the interviewer can review security-related activity.

A. *Secure Coding Monitoring*

When secure mode is active, the application records selected interactions with the coding page. These include tab changes, loss of window focus, leaving fullscreen mode, clipboard operations, keyboard shortcuts, right-click actions, navigation, and periods of inactivity. Each recorded event is associated with the authenticated interview session and a timestamp.

Browser visibility and focus events are used to detect when the participant moves away from the coding page. The application can also detect a change in fullscreen state and issue a warning when configured to do so. Clipboard and keyboard-related actions can be assigned policies such as allow, warn, or block. These controls are intended to provide additional visibility during an assessment, rather than claiming that browser monitoring can detect every form of outside assistance.

B. *Violation and Warning Management*

Security events are passed to a central violation-management component. Depending on the interview configuration, an event can be recorded with a severity ranging from informational to critical. Repeated events can increase the violation count and may lead to an action such as a warning, temporary locking of the session, or termination.

The participant can receive an on-screen warning when a configured event occurs, while the interviewer receives the corresponding notification. For instance, leaving the coding window or exiting fullscreen can be recorded and displayed immediately. The system therefore presents the events as evidence for the interviewer to review instead of treating one event by itself as proof of misconduct.

C. *Session and Anti-Tampering Protection*

The interview mode also checks that requests originate from an authenticated participant belonging to the correct

ISSN: 3107-6513

interview room. A session is associated with the participant, interview, room, and active-session identifiers. Before a security-sensitive operation is accepted, the server checks membership, permissions, and the current session state.

Values that affect the security state are kept on the server. This includes violation counts, lock status, participant roles, and session permissions, rather than allowing the browser to decide these values. The interview configuration can also restrict simultaneous sessions and allow an interviewer to revoke an unauthorized or duplicate session.

D. Real-Time Security Events and Audit Logging

The same Socket.IO connection mechanism is used to deliver security notifications to the interviewer. Events such as tab changes, fullscreen exits, clipboard attempts, session changes, and warnings are sent through the authenticated session channel.

Each interview session has its own security-event log. The stored record contains identifiers for the interview, participant, and session together with the event type, severity, time, and related metadata. The resulting log gives the interviewer a chronological record that can be reviewed after or during the assessment.

E. Interviewer Security Dashboard

A dedicated dashboard gives the interviewer a current view of the participant's session. It shows the session state, violation count, fullscreen status, recent security events, and other session details. Socket.IO updates the event information as new events are generated, allowing the interviewer to examine activity while the assessment is in progress.

The dashboard deliberately reports individual events rather than automatically deciding that a participant has cheated. The interviewer can therefore consider the event history and the circumstances of the complete session before drawing a conclusion.

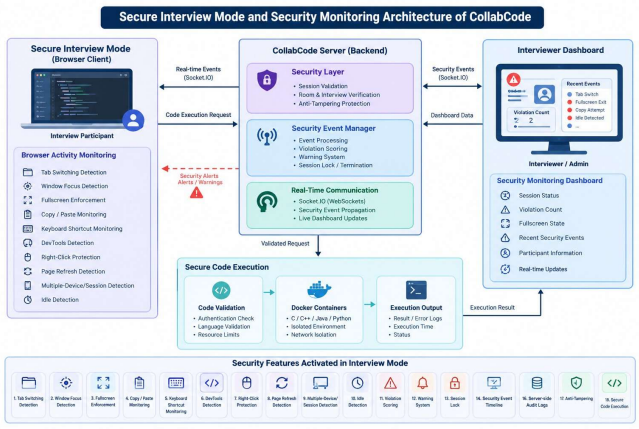

Fig. 3. Secure Interview Mode and Security Monitoring Architecture of CollabCode.

F. Secure Code Execution

Secure-mode execution uses the same server-side execution service as normal collaborative coding, but additional checks are performed first. The backend verifies authentication, interview membership, room permissions, and lock status. It also checks the submitted code size, selected language, execution frequency, and output limits before forwarding the request.

The submitted program continues to run inside the isolated container environment. Keeping execution separate from the main application reduces the possibility of user code directly interacting with the application server and allows resource and execution limits to be enforced during both normal collaboration and interviews.

V. Result and Analysis

The evaluation covered four usage cases: editing by one user, two users working together, rooms with several participants, and sessions in which AI assistance was enabled. For these cases, the measured quantities included synchronization delay, program execution time, delivery reliability, and the number of simultaneous users supported.

A. Synchronization Performance

The synchronization measurements are listed in Table I. The observed latency was lower than the HTTP-polling baseline used in the comparison, with the overall reduction reported as 30% [6]. This follows the general advantage of persistent WebSocket communication reported in related collaborative systems [10].

TABLE I. Synchronization Performance Across Scenarios

Scenario	Avg Latency	Reliability	Users
Single User	<20 ms	99.9%	1
Two Users	<35 ms	99.5%	2
Multi-User	<50 ms	98.8%	5–10
Peak Load	<80 ms	97.5%	10+

B. Code Execution Performance

Execution measurements for the four supported languages are given in Table II. The recorded average response time remained below 1.5 seconds for every language in the test. The programs were run in isolated Docker environments so that one session did not share its execution environment with another [4].

TABLE II. Code Execution Response Time per Language

Language	Avg Response Time	Accuracy
Python	0.8 s	99.2%
Java	1.3 s	98.7%
C	0.9 s	99.0%
C++	1.0 s	98.9%

C. Overall Platform Performance

The combined measurements are shown in Table III. In the recorded tests, AI suggestions were delivered within 800 ms with a reported reliability of 96%. This result is presented alongside the synchronization, execution, and chat measurements rather than treating the AI component as part of the real-time editing path. Related work on LLM-assisted development is reported by Rasheed et al. [16].

The comparison in Fig. 4 brings together the main measured indicators: synchronization delay, execution accuracy, and reliability of AI assistance. It provides a visual summary of the values reported in the evaluation tables.

TABLE III. OVERALL PERFORMANCE ACROSS COLLABCODE MODULES

Module	Reliability	Latency
Real-Time Sync	99.5%	<35 ms
Code Execution	98.9%	<1.5 s
AI Code Assistance	96.0%	<800 ms
Real-Time Chat	99.8%	<20 ms
Full Platform	98.5%	<50 ms

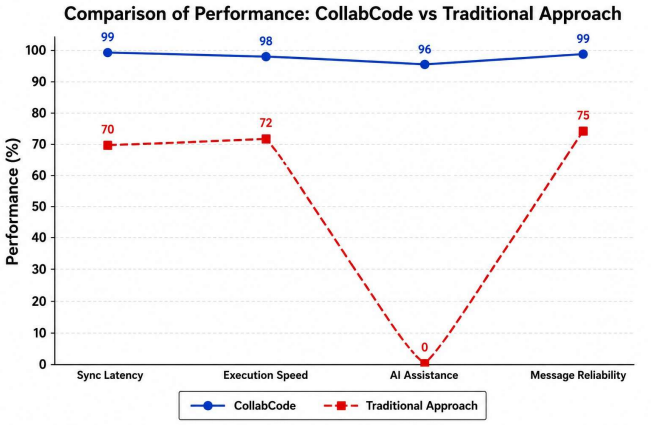

Fig. 4. Performance Comparison: CollabCode vs Traditional Approaches.

VI. CONCLUSION AND FUTURE SCOPE

The implemented CollabCode platform brings shared editing, server-side program execution, AI assistance, persistent rooms, and a Secure Interview Mode into one browser-based system. Socket.IO handles the live collaboration path, Docker separates program execution from the main server, and MongoDB keeps the information required for room recovery. The interview component adds browser-event logging and server-side session checks without applying those restrictions to ordinary rooms.

In the reported experiments, synchronization stayed below 50 ms for rooms with up to ten concurrent users, and program execution remained below 1.5 seconds. Message delivery reliability was above 98%. AI suggestions were handled separately from the synchronization path, while Secure Interview Mode recorded events such as tab changes, focus changes, fullscreen state, clipboard activity, and session changes. The resulting event history can be reviewed by the interviewer as part of the assessment.

Several extensions remain possible. CRDT-based synchronization could be investigated for rooms with larger numbers of simultaneous editors [8]. Integration with GitHub or GitLab would connect live sessions with conventional version control. Other planned directions include video and screen sharing, multi-file workspaces, stronger AI-based bug detection and review [17], [18], mobile clients, continuous integration and continuous deployment (CI/CD) support, enterprise authentication with role-based access control, richer security analytics, and synchronization that can tolerate intermittent connectivity.

ACKNOWLEDGMENT

The authors thank the Department of Information Science & Engineering, CMR Institute of Technology, Bengaluru, for providing the necessary resources and academic environment to support this research.

REFERENCES

[1] K. Virdi, A. L. Yadav, A. A. Gadoo, and N. S. Talwandi, "Collaborative code editors—enabling real-time multi-user coding and knowledge sharing," in *2023 3rd International Conference on Innovative Mechanisms for Industry Applications (ICIMIA)*, 2023, pp. 614–619.

[2] D. Sun and C. Sun, "Operational Transformation Algorithms for Supporting Group Undo," *Journal of Systems and Software*, vol. 80, no. 4, pp. 562–574, 2007.

[3] A. Dhawan, A. Singh, C. Gupta, P. Goel, and S. Teotia, "Collaborative Landscape of Software Development: RTC Code Editor," *International Research Journal on Advanced Science Hub*, vol. 6, no. 5, 2024.

[4] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[5] M. Chen, J. Tworek, H. Jun *et al.*, "Evaluating Large Language Models Trained on Code," *arXiv preprint arXiv:2107.03374*, 2021.

[6] V. Pimentel and B. G. Nickerson, "Communicating and Displaying Real-Time Data with WebSocket," *IEEE Internet Computing*, vol. 16, no. 4, pp. 45–53, 2012.

[7] C. Xu, P. Liang, and X. Li, "Real-time collaborative programming in distributed systems: A framework perspective," in *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2023, pp. 456–460.

[8] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-Free Replicated Data Types," in *Proc. Symposium on Self-Stabilizing Systems (SSS)*, Grenoble, France, 2011, pp. 386–400.

[9] M. Zhang, Y. Jin, J. Xu, and Z. Yang, "Intelligent conflict resolution in collaborative programming environments," *Journal of Software Engineering Research and Development*, vol. 14, no. 2, pp. 113–129, 2023.

[10] H. Fan, K. Li, X. Li, T. Song, W. Zhang, Y. Shi, and B. Du, "CoVSCode: a novel real-time collaborative programming environment for lightweight IDE," *Applied Sciences*, vol. 9, no. 21, p. 4642, 2019.

[11] H. Fan, H. Zhu, Q. Liu, Y. Shi, and C. Sun, "Shared-locking for semantic conflict prevention in real-time collaborative programming," in *2017 IEEE 21st International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, 2017, pp. 174–179.

[12] T. Nakamura, H. Kinoshita, and A. Suzuki, "A semantic approach to prevent merge conflicts in collaborative IDEs," in *Proc. 2022 International Conference on Intelligent Computing (ICIC)*, 2022, pp. 78–85.

[13] Microsoft Corporation, "Monaco Editor: The Code Editor that Powers Visual Studio Code," GitHub Repository, 2024. [Online]. Available: https://github.com/microsoft/monaco-editor

[14] A. Y. Wang, Z. Wu, C. Brooks, and S. Oney, "Don't Step on My Toes: Resolving Editing Conflicts in Real-Time Collaboration in Computational Notebooks," *arXiv preprint arXiv:2404.04695*, 2024.

[15] J. Park, S. Lee, and H. Jung, "Live deployment in real-time collaborative programming tools," in *Proc. 2023 IEEE International Conference on Cloud Engineering (IC2E)*, 2023, pp. 312–317.

[16] Z. Rasheed, M. Namoun, A. Mehandjiev, A. Tufail, and K.-H. Lam, "AI-Powered Code Review with Large Language Models: Early Results," *arXiv preprint arXiv:2404.18496*, 2024.

[17] M. Kawalerowicz, M. Pietranik, and K. Stepniak, "LLMs as Code Review Agents: A Rapid Review and Experimental Evaluation with Human Expert Judges," in *Computational Collective Intelligence. ICCCI 2025*, Lecture Notes in Computer Science, vol. 16138, Springer, 2025.

[18] S. Sridhara, G. Gao, and D. Chandra, "AICodeReview: Advancing code quality with AI-enhanced reviews," *SoftwareX*, vol. 26, 2024.

[19] D. Kulikov, E. Shumilova, and P. Smirnov, "Enhancing code sharing and integration in distributed development," in *Proc. 2023 European Conference on Software Architecture (ECSA)*, 2023, pp. 190–198.

[20] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *ACM Queue*, vol. 14, pp. 70–93, 2016.

[21] K. Chodorow, *MongoDB: The Definitive Guide*, 3rd ed. Sebastopol, CA: O'Reilly Media, 2019.