IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 3, March 2026

DOI: 10.17148/IJARCCE.2026.15304

Automate Infrastructure using AWS CloudFormation and Docker

S SREEKA[1], Dr. S. SHYLAJA[2]

Student, Department of Computer Applications, Sri Ramakrishna College of Arts & Science (Autonomous),

Coimbatore – 641006, Tamil Nadu, India[1]

Assistant Professor, Department of Computer Applications, Sri Ramakrishna College of Arts & Science (Autonomous),

Coimbatore – 641006, Tamil Nadu, India[2]

Abstract: Manual cloud infrastructure management leads to environment inconsistencies, deployment errors, and significant operational overhead. Existing productivity timer applications suffer from poor visual engagement and lack modern deployment practices. This paper presents Focus Timer, a React 19 web application with a glassmorphic interface, adaptive time-of-day theming, and SVG-based circular progress visualization, deployed to AWS ECS Fargate through a fully automated pipeline using Docker and AWS CloudFormation.

The multi-stage Docker build produces a production image of approximately 25 MB. The Vite 7 build generates a gzipped bundle under 50 KB with sub-second page load. CloudFormation templates reproducibly provision an ECS Cluster, IAM Execution Role, and Fargate Task Definition across any AWS region. Automated PowerShell scripts complete the full pipeline from source to running cloud service in under six minutes, eliminating all manual console steps. All 20 functional test cases passed across Chrome, Firefox, Edge, and Safari.

Keywords: React 19, Vite 7, AWS CloudFormation, Docker, ECS Fargate, Infrastructure-as-Code, Glassmorphism, SVG Animations, ECR.

I. INTRODUCTION

Cloud computing has transformed how organizations deploy web applications, offering elastic scalability and global availability. Yet a significant gap remains between the simplicity of building a modern frontend and the complexity of deploying it reliably. Manual configuration through the AWS Management Console is error-prone, unreproducible, and impossible to version-control. Infrastructure-as-Code using AWS CloudFormation solves this by defining resources declaratively in YAML, enabling consistent, auditable deployments across environments.

Containerization using Docker addresses a complementary problem: ensuring an application behaves identically regardless of where it runs. Multi-stage builds separate the build environment from the production image, reducing final image size by over 90% and eliminating development dependencies from production. This project combines both practices to deploy Focus Timer, a React 19 single-page application, to AWS ECS Fargate through a fully scripted pipeline. The application demonstrates advanced frontend patterns React hooks, CSS glassmorphism, SVG animations, and adaptive theming while the deployment pipeline demonstrates production-grade DevOps practices applicable to any frontend application.

II. OBJECTIVE

This project aims to design a production-quality React web application and demonstrate its deployment through a fully automated Infrastructure-as-Code pipeline. Specific goals include:

- Develop a React 19 + Vite 7 application with SVG animations, CSS glassmorphism, and adaptive time-of-day theming.
- Containerize the application using a multi-stage Docker build producing a minimal, secure production image.
- Provision AWS infrastructure reproducibly using CloudFormation templates with an ECS Cluster, IAM Role, and Fargate Task Definition.
- Automate the complete pipeline from Docker build to CloudFormation stack management through scripted deployment.
- Evaluate build performance, image size, deployment timing, and cross-browser compatibility.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 3, March 2026

DOI: 10.17148/IJARCCE.2026.15304

III. EXISTING SYSTEM

Current deployment practices and timer tools share well-documented limitations. Manual AWS console deployments are not version-controlled and cannot be reliably reproduced when infrastructure must be recreated, the entire manual process repeats with no consistency guarantee. Single-stage Docker builds include Node.js, npm, and development dependencies in the production image, inflating image sizes to over 300 MB unnecessarily. Without containerization, environment-specific dependency conflicts frequently cause production failures that do not occur during development.

Existing timer applications, physical timers, OS alarm apps, browser-based Pomodoro tools provide only basic digital countdowns with no visual progress indication, no adaptive theming, and no inline editing. None are containerized or deployable through standardized DevOps pipelines. Table 1 summarizes these limitations.

Category	Existing Approach	Limitation
Infrastructure	Manual AWS console config	Not reproducible, no version control
Containerization	Single-stage Docker builds	Large images with dev dependencies (>300 MB)
Timer UI	Basic digital countdown	No visual progress, no adaptive theming
Automation	No scripted pipeline	Manual steps required on every deploy

Table 1: Limitations of Existing Systems

IV. PROPOSED SYSTEM

4.1 React Application Architecture

Focus Timer is a zero-backend single-page application built with React 19 and Vite 7. All logic runs client-side through a single App component. State is managed via seven useState variables: timeLeft and totalTime (seconds), isActive (boolean), isEditing and editValue (inline editor), currentTime (Date), and timeOfDay (string key). Two useRef variables hold the DOM reference for the inline editor and the setInterval ID for cleanup.

Four useEffect hooks drive the application's behavior: time-of-day detection on mount, a live clock interval updating currentTime every second, the countdown engine that decrements timeLeft each second when isActive is true and auto-stops at zero, and an auto-focus effect for the inline editor. All intervals return cleanup functions to prevent memory leaks on unmount.

4.2 User Interface Features

The glassmorphic card uses backdrop-filter: blur(20px) over time-of-day background images, adapting to four periods shown in Table 2. A 120×120px SVG ring uses stroke-dashoffset with a 1s linear CSS transition to smoothly deplete as time elapses. Six inline SVG components (PlayIcon, PauseIcon, CoffeeIcon, SunIcon, SunsetIcon, MoonIcon) eliminate external icon library dependencies. Preset buttons (5, 10, 20 minutes) are disabled during active countdown. Custom minute/second inputs cap seconds at 59 via Math.min. The inline editor activates on click when paused, restricts input to four digits, and parses MMSS format on Enter or blur.

Period	Hour Range	Icon	Greeting
Morning	5 AM – 11:59 AM	Coffee ☕	Good Morning
Afternoon	12 PM – 4:59 PM	Sun ☀	Good Afternoon
Evening	5 PM – 8:59 PM	Sunset □□	Good Evening
Night	9 PM – 4:59 AM	Moon □□	Good Night

Table 2: Time-of-Day Theme Configuration

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 3, March 2026

DOI: 10.17148/IJARCCE.2026.15304

4.3 Deployment Architecture

The Dockerfile uses a two-stage build: Stage 1 (node:18-alpine) installs dependencies and runs npm run build; Stage 2 (nginx:alpine) serves the compiled dist/ assets. The nginx.conf applies a try_files $uri $uri/ /index.html SPA fallback and enables gzip compression. The infrastructure.yaml CloudFormation template provisions three resources: an ECS Cluster, an IAM Task Execution Role (AmazonECSTaskExecutionRolePolicy), and a Fargate Task Definition (256 CPU / 512 MB, awsvpc network, port 80). Deployment scripts handle ECR authentication, image push, and CloudFormation stack create-or-update with wait commands. Figure 1 shows the complete pipeline.

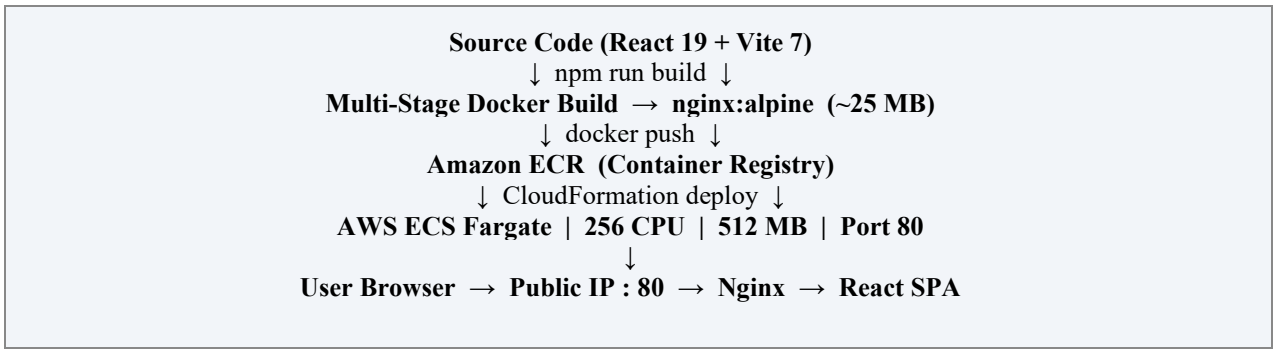

Figure 1: End-to-End Deployment Architecture

V. METHODOLOGY

Development followed five structured phases. Phase 1 scaffolded the project using npx create-vite@latest with the React template, configuring ESLint with react-hooks and react-refresh plugins, and setting up .dockerignore to exclude node_modules and dist/ from the build context. Phase 2 developed the React application incrementally first implementing core state and the countdown engine, then adding preset buttons and custom inputs, and finally building the glassmorphic UI, SVG progress ring, time-of-day theming, and inline editing. The handleEditSubmit function parses a four-digit MMSS string, extracting minutes from the leading digits and seconds from the last two, capped at 5999 total seconds via Math.min.

Phase 3 containerized the application using the multi-stage Dockerfile, placing COPY package*.json./ before COPY . . to maximize Docker layer caching. Phase 4 defined infrastructure as code in the CloudFormation YAML template, using !GetAtt to create implicit resource ordering between the Task Definition and IAM Role, and specifying CAPABILITY_NAMED_IAM during deployment. Phase 5 performed functional testing across 20 timer scenarios, visual testing for glassmorphic effects and responsive layout, build testing for all npm and Docker commands, and browser compatibility testing across Chrome, Firefox, Edge, and Safari.

VI. RESULT AND DISCUSSION

6.1 Application and Build Performance

All performance targets were met or exceeded. The Vite production build produced a JavaScript bundle of approximately 142 KB minified and 48 KB gzipped, well below the 500 KB target. Combined CSS totalled approximately 12 KB. Initial page load times measured under 800 ms on standard broadband. The 1-second countdown interval triggered React reconciliation cycles of approximately 2 ms, comfortably within the 16 ms frame budget for 60 fps rendering. The multi-stage Docker build produced a final image of approximately 25 MB compared to over 300 MB for a comparable single-stage build—a reduction of over 90%.

Metric	Target	Actual	Status
Initial page load	< 2 s	~800 ms	Pass
JS bundle (gzipped)	< 500 KB	~48 KB	Pass
Timer update latency	< 16 ms	~2 ms	Pass
Docker image size	< 50 MB	~25 MB	Pass
Browser memory usage	< 50 MB	~30 MB	Pass

Table 3: Performance Test Results

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 3, March 2026

DOI: 10.17148/IJARCCE.2026.15304

6.2 Deployment Pipeline Results

The deployment pipeline executed successfully across multiple test runs. The Docker build completed in approximately 90 seconds on first run and under 30 seconds on subsequent runs due to layer caching. ECR push transferred approximately 25 MB in under 60 seconds. CloudFormation stack creation completed in approximately 3 minutes; updates in under 2 minutes. Fargate tasks transitioned from PROVISIONING to RUNNING in approximately 45 seconds. Table 4 summarizes deployment timings.

Step	First Run	Repeat Run	Status
Docker build	~90 s	~30 s	Pass
ECR image push	~60 s	~15 s*	Pass
CloudFormation create	~3 min	N/A	Pass
CloudFormation update	N/A	~2 min	Pass
Fargate task RUNNING	~45 s	~45 s	Pass

Table 4: Deployment Pipeline Timing (* cached ECR layers)

6.3 Functional and Compatibility Testing

All 20 functional test cases passed. Preset buttons correctly initialize totalTime and timeLeft to 300, 600, and 1200 seconds and set isActive to true. Custom inputs correctly cap seconds at 59 and update both state variables synchronously. The inline editor correctly filters non-digit input, limits length to four characters, and parses MMSS on Enter or blur. The Stop button resets all states to defaults. The automated stop at timeLeft === 0 was confirmed across all preset and custom durations. Browser compatibility was confirmed across Chrome 120+, Firefox 115+, Edge 120+, and Safari 17+, with the -webkit-backdrop-filter prefix handling Safari's glassmorphism requirement.

VII. CONCLUSION

This paper presented Focus Timer, a React 19 web application with a glassmorphic interface, time-of-day adaptive theming, and SVG progress visualization, deployed to AWS ECS Fargate through a fully automated CloudFormation and Docker pipeline. Experimental results confirm a 90% reduction in Docker image size through multi-stage builds, sub-second page load times from Vite's optimized output, and a complete automated deployment in under six minutes with zero manual steps. All functional and compatibility tests passed.

The proposed approach demonstrates that Infrastructure-as-Code and containerization offer significant benefits even for small-scale frontend applications, providing reproducibility, version control, and deployment speed that manual approaches cannot match.

VIII. SCOPE FOR FUTURE ENHANCEMENT

The Focus Timer application, while fully functional in its current form, offers substantial scope for further development across several dimensions.

- Audio and Visual Notifications: The current implementation silently reaches zero with no alert. Integrating the Web Audio API for customizable completion sounds and the browser Notification API for system-level alerts would significantly improve usability, especially when users are working in another tab.
- Session History and Analytics: Leveraging the browser's localStorage API to persist completed session data across reloads would enable users to track total focus time, sessions completed, and daily streaks, visualized through a charting library such as Chart.js.
- Pomodoro Mode: Implementing automated work and break cycles following the 25-minute work and 5-minute break pattern would enable structured productivity workflows using a React state machine to manage transitions between session types.
- Progressive Web App (PWA) Support: Introducing a service worker via Workbox would enable offline functionality, background sync, and home-screen installation on mobile devices, extending the application's reach beyond the browser.

International Journal of Advanced Research in Computer and Communication Engineering
Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 3, March 2026
DOI: 10.17148/IJARCCE.2026.15304

- CI/CD Pipeline: Integrating GitHub Actions to automate testing, Docker image building, ECR pushing, and CloudFormation stack updates on every push to the main branch would ensure consistent and reliable releases without manual script execution.
- User Authentication and Cross-Device Sync: Adding OAuth 2.0 login via Google or GitHub through Firebase Auth would allow users to synchronize settings, preferences, and session history across multiple devices.
- Elastic IP and Custom Domain: Provisioning an AWS Elastic IP, Route 53 hosted zone, and ACM SSL certificate would provide a persistent HTTPS URL, eliminating the need to look up a new public IP each time a Fargate task is created.
- Ambient Sounds: Offering background audio options such as rain, café noise, or lo-fi music during focus sessions using the Web Audio API or Howler.js would create a more immersive focus environment.
- Keyboard Shortcuts: Adding hotkey support—Space for play/pause, R for reset, and number keys 1/2/3 for presets—through React keyboard event handlers would improve accessibility and efficiency for power users.
- Team Timers: Introducing real-time synchronized timers via WebSocket using Socket.io would support pair programming, group study sessions, and team-based Pomodoro sprints across distributed teams.

REFERENCES

[1]. Facebook Inc. (2024). React: A JavaScript library for building user interfaces. https://react.dev/
[2]. Evan You & Vite Contributors (2024). Vite: Next Generation Frontend Tooling. https://vitejs.dev/
[3]. Amazon Web Services (2024). AWS CloudFormation User Guide. https://docs.aws.amazon.com/cloudformation/
[4]. Docker Inc. (2024). Docker Documentation: Multi-stage builds. https://docs.docker.com/build/building/multi-stage/
[5]. Amazon Web Services (2024). Amazon ECS Developer Guide: AWS Fargate. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/AWS_Fargate.html
[6]. Mozilla Developer Network (2024). Web APIs: SVG, CSS Animations, Date API. https://developer.mozilla.org/
[7]. Kim, G., Humble, J., Debois, P., & Willis, J. (2021). The DevOps Handbook. IT Revolution Press, 2nd Edition.

