CI/CD PIPELINE SIMULATOR: A BROWSER-BASED FRAMEWORK FOR INTERACTIVE DEVOPS EDUCATION AND DEPLOYMENT WORKFLOW VISUALIZATION

Dr. Basanti Ghanti[1], Kajal Gurung[2]

Access Online

Article DOI:
10.5281/zenodo.21410124

[1] Professor, Electronics and Communication Engineering Faculty of Engineering and Technology Exclusively for Women, Sharnbasva University
[2] Student, Electronics and Communication Engineering Faculty of Engineering and Technology Exclusively for Women, Sharnbasva University

How to Cite:
Dr. Basanti Ghanti & Kajal Gurung (2026), CI/CD Pipeline Simulator: A Browser-Based Framework for Interactive DevOps Education and Deployment Workflow Visualization, International Educational Journal of Science & Engineering (IEJSE),Vol: 9, Special Issue, May 2026 224-227

ABSTRACT

The complexity of modern software delivery has proven that CI/CD pipelines are crucial elements of professional practices. Despite its wide application in the real world, practical tools to learn and explore the concept remain limited in educational context. This paper proposes a desktop-based CI/CD Pipeline Simulator built using Tkinter, to bridge theoretical concepts and practical execution. The simulation consists of the three main parts of the deployment pipeline, build, test and deployment. The integrated tool runs all three parts sequentially within a single executable. A user can input Python code, then the system can compile and scan code for possible errors, and simulates the deployment process. Real-time progress bars and a log to detail the step by step processing provide users instant feedback throughout the process. Upon successful deployment, it creates a dynamic simulated prototype of an application, allowing UI characteristics such as labels on buttons, colors of backgrounds, and interactive functions to be changed by only manipulating variables within the code. It encourages learners to experience pipeline steps without needing to configure servers, cloud accounts, or complex tooling. Experiments suggest the tool effectively demonstrates basic DevOps practices such as verification of build, conditional testing gate and environmental dependent deployment. The tool is portable across different platforms, and effective for undergraduate courses, independent study, and rapid prototyping. This project establishes a feasible educational tool for DevOps teaching, and laid groundwork for further developments such as container integration, trigger of branches and multi-environment staging

KEYWORDS: CI/CD pipeline, DevOps Education, Tkinter, Software Deployment Simulation, Continuous Integration, Pipeline Visualization, Python GUI

1. INTRODUCTION

Coding teams across the place are picking up DevOps habits quickly, trying to push releases faster, reduce bugs during merges, yet hold quality firm. At the core lives the CI/CD pipeline - a chain of automated stages shifting fresh code into tested, deploy-ready changes. Though solid platforms such as Jenkins, GitHub Actions, or GitLab CI deliver these pipelines in actual projects, setting them up usually seems messy, creating hurdles for solo testing or teaching settings.

Imagine understanding all the rules of a game without ever playing. That is how it feels when someone talks confidently about CI/CD but freezes at the first failed build. Moments earlier, things seemed clear. Then a cascade of test failures breaks the rhythm. Knowledge exists, yes - still, practice stumbles. Watching automation run live? Rare. Most have only read about it. The gap isn't theory. It's doing. Pages list actions step by step, flat and lifeless. Drills in school pull pieces apart until nothing feels whole - neat boxes that ignore messy truth. When a live system runs, trouble spreads fast - one hiccup freezes everything, pressure builds quick. Errors on display sink deeper than quiet tests ever could.

This project shows a hands-on system made with Python and Tkinter, designed like a real

Research Paper

CI/CD setup. Rather than clicking through vague phases, users write live code right where they see it, sending it step by step into a pretend deployment chain. Since every phase operates straight from the machine, no web connection or remote machines are needed at any point. Progress lights up moment by moment - revealing whether each build lives or dies depending on the lines just typed. A working model wakes up once the last step finishes, built entirely from what lives in the code. Because edits change how things look, checking them acts less like reading reports and more like sketching with rules. Most systems split writing code from seeing output - this one shows both at once, walking hand in hand. Tiny adjustments reshape results instantly, making mistakes and fixes appear clear as day. Each nudge writes its own consequence onto the screen.

Something grabs attention straight off - a lean simulator that mimics CI/CD flow step by step, uses little power, works alone, carries its own world inside. Beyond that, tweaking the look happens directly in code, where shifts show up instantly, clear to see, no waiting. Here's another angle: tools like this for grasping DevOps concepts link up easily, demand almost nothing extra, slide into different environments without noise. Next unfolds what waits down the path: chapter two traces how it moves and clicks, along with reasons behind each pick; chapter three unpacks layout and assembly logic; chapter four stacks performance numbers beside peers; after that, nods go out, references follow, silence settles.

2. EASE OF USE

A. Accessibility and Zero-Configuration Design

One of the design goals of the CI/CD Pipeline Simulator was to eliminate all common hurdles associated with pipeline tools. In conventional CI/CD platforms, setting up any given pipeline is not an insignificant undertaking; users must first acquire a server and set all environment variables, configure YAML files, and establish authentication tokens, only then to attempt to run a single pipeline. For students just learning the fundamentals, it is extremely de-motivating and confusing. The pipeline was specifically designed to not require anything beyond a standard installation of Python. Users do not need to install any additional libraries, need not configure any files, nor connect to any remote services. An entire application is kicked off by a single Python script, resulting in a user interface that is ready to use the moment it loads. The need to install nothing, configure nothing, and log in to nothing was chosen based on the firm belief that educational tools should reduce the friction to get started, not add to it. The UI follows a vertical, flow-based layout which accurately simulates the pipeline stages in order. The code input section is at the top, the action button below it, followed by the logs, then the mock server and application viewer. This allows students to build an accurate mental model of how their changes progress through the pipeline stages, reinforces a consistent visual metaphor. Color coded status indications, along with a real-time progress bar reduce cognitive load by providing unambiguous, at-a-glance feedback.

B. Maintaining the Integrity of the Specifications

Something shifts without warning - understanding clicks into place like it was always there. Real code runs the system now, so when new text drops in, differences show up immediately. Change the label_text field, watch every word shift on its own. Tweak app_bg, colors respond exactly where they rest. Out of nowhere, code appears - suddenly things start moving differently. Each tweak becomes visible at once, no delays, nothing hidden. Pictures form straight from words on screen, clear and fast. Walking forward slowly, ideas begin making sense - small edits twist the whole picture. Watch closely now, there it is, code showing itself openly. Out of nowhere, everything sharpens. The hazy idea becomes real right there.

A lone strange symbol, placed out of place, stops everything before running. Right when checking begins, issues show up fast, no hiding them. Even if someone just says "mistake," nothing changes - tests still fail, release gets blocked automatically. People see real protection happening, not guesses about safety. Changes apply themselves. Machines repair without help, showing progress moves in loops - response then small fix. Outcomes that show up quick make lessons stick around longer. Studies back it: doing something beats just listening, every time. When tests link chunks of code into a single path, they act like checkpoints. If quality slips even a bit, progress hits a wall. Speed matters at first, yet thorough work holds weight after ability starts building.

Halfway up, the bar stops moving. If validation trips, everything resets fast. Warnings show, color fades, actions lock down. One check examines shape, while a different one tests behavior - proof of separate jobs done. Reversal hits hard when steps misfire. Just because something looks fine doesn't mean it works once you try. Letters follow one another, quiet and constant, like air shaping moments.

3. SYSTEM ARCHITECTURE AND IMPLEMENTATION

A. Build stage Implementation

Clicking the second Run Pipeline pulls everything from the coding section. Out it goes, straight into Python's compile() function - this step checks if things follow proper structure rules. Syntax errors show themselves here, way

ahead of any actual run. Once things start moving, this stage acts like a checkpoint. Instead of running anything, it just looks at how the pieces fit together.

Midway through, after the compiler finishes without errors, a time-stamped line pops into view while the bar moves up to thirty percent. Then everything pauses briefly - controlled by an internal delay - just so the confirmation can sink in before tests launch. When something goes wrong during compilation, the system logs a warning, shifts the indicator to red, then stops every operation on the spot. Much like real-world builds, any further steps stay locked out if problems show up.

B. Test Stage and Quality Gate Logic

When the build completes smoothly, attention shifts straight to testing. Not just watching how things run, but digging into what the code actually states. Each chunk of text gets scanned specifically for the term "error," hunting down glitches - whether placed there or slipped in. Real checks often dive far below this surface, yet this captures the core thought clearly enough. Getting through means beating that one clear condition - there is no bending it.

Most of the bar turns solid when checks pass, though rollout still lines up behind. A single failure dumps it all - color snaps to red, every process freezes. Building works does not mean testing agrees - one proves structure, the other reveals collapse under pressure. Text scrolls as if alive, each character arriving just after the last, heartbeat timing giving weight to seconds passing.

C. Deployment Stage and Runtime Execution

At this point, the code written in Python will be executed using exec(), where all variables will be kept separate from each other. This process will take place after all other previous actions have been taken care of in the sequence. In order to capture the result, there is also a separate handler that redirects all display results into the test server window instead of going to the hidden terminal. Once the process has finished executing the code, certain variable values set in the process will be evaluated. These affect what can be seen in the visual preview below. For example, if the variable called label_text is given the value "Hello World", then this string will be used as the text at the top. Button_label takes button_text as input and displays it as the label of the main button. The background of the layout as a whole is given by the variable called app_bg, and so on. A second button will also be displayed if the Boolean variable show_second_button is set to True; its label comes from the variable second_button_text. All changes made are reflected immediately.

D. GUI Layout and User Interface Components

Fresh start on design focuses on look, built only with Python's Tkinter - needs nothing more, works in small setups. At the top: the header, clean and steady. Below opens space for typing code yourself. Then comes a button to begin actions, placed right over message lines showing progress during jobs. Down below, you'll find a window displaying replies directly from the server. Ending finally within a live frame - this is where the active tool becomes visible.

Silence fills the layout, soaked in deep blue, framed by heavy gray strips standing firm like secured data units. Lines of text line up straight and tight, cut clean, mirroring old terminal screens long out of use. Instead of blending, corners pull back - gaps form shape, give room for pieces to stand separate. Out of stillness, a thin marker creeps beside the edge, set through ttk.Progressbar, tracking steps not by beeps but shifts taken. Light jumps blue when things shift within. Finished jobs bring a quiet green instead. If trouble strikes, red slices through without warning. What you see is what it means - clear at first glance. No sound breaks the air, still each color speaks loud. One step at a time, patterns start making things clear. Because of the room between parts, meaning begins to form. Built right into its core, simplicity stays put.

4. RESULTS AND EVALUATION

Testing showed that the build step reliably caught and reported all syntax issues in Python code, such as missing parentheses, improper indentation, or syntax structures. In each case, the whole process got stuck at the build step, and the relevant error message popped up, halting any further progression. At the same time, code without any syntax issues automatically passed this stage, regardless of their semantic correctness, which is consistent with the standard behavior in compilation practice. As for the test step, it worked properly, stopping the deployment process if the word "error" was used anywhere in the code. It did not matter whether the mentioned word was an identifier name (such as error_code = 404), part of a comment, or even appeared twice: the system analyzed the source text directly. After passing both steps, the application would deploy, and the prototype application would get updated based on the variables of the inputted code.

A single misplaced bracket brought everything to a stop - each test showed the system pouncing on tiny errors without waiting. It wasn't only gaps between lines; strange alignment alone could lock things up fast, dragging hidden bugs into view immediately. Progress resumed solely once structure fell neatly together, acting nearly identical to real compilation software. Each mistake triggered an immediate pause, matched with a clear

explanation, every oversight caught before moving on. Approval came through shape, never reason. Every time, the outcome stayed fixed: tangled scripts stalled while clean ones moved ahead.

Something moved forward each time - until a word showed up and everything stopped. When that label popped into view, progress died on the spot. Even a quiet mention, say error_code = 404 tucked within comments, set off alarms. It seems phrases are checked before any layout is verified. To win through, both steps had to clear together; only then did anything finish. Afterward, it shifted just right - guided only by fresh figures in the code. Each time, that rhythm held firm.

Fig. 1.1

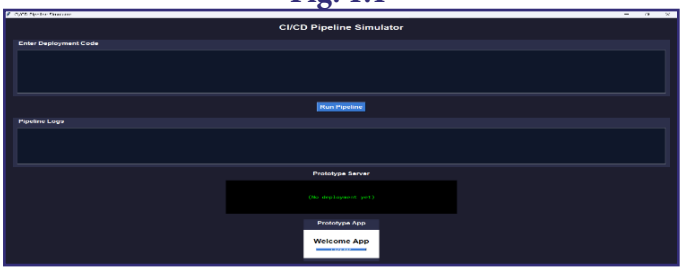


Fig. 1.2

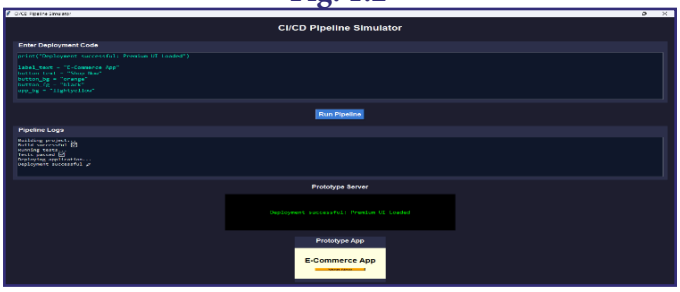
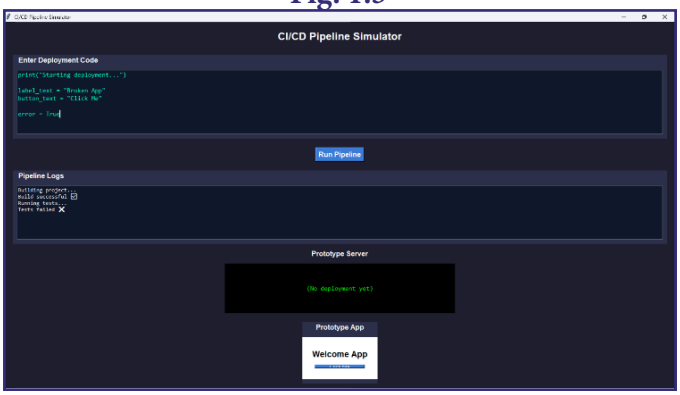


Fig. 1.3



ACKNOWLEDGMENT

Thanks go to the teachers and classmates who shared thoughts while we built and tested this project. Because of their comments, the way it works and teaches got much sharper. The people who write code in Python and share how they do it helped a lot too - finding answers became easier. Money did not come from outside groups to pay for any part of this study.

REFERENCES

1. M. Shahin, M. A. Babar, and L. Zhu, “Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices,” IEEE Access, vol. 5, pp. 3909–3943, 2017.
2. L. Chen, “Continuous delivery: Huge benefits, but challenges too,” IEEE Software, vol. 32, no. 2, pp. 50–54, Mar. 2015.
3. J. Humble and D. Farley, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Boston, MA: Addison-Wesley, 2010.
4. B. Fitzgerald and K. J. Stol, “Continuous software engineering: A roadmap and agenda,” J. Syst. Softw., vol. 123, pp. 176–189, Jan. 2017.
5. S. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora, and M. Di Penta, “How open source projects use static code analysis tools in continuous integration pipelines,” in Proc. IEEE/ACM 14th Int. Conf. Mining Softw. Repositories (MSR), Buenos Aires, Argentina, 2017, pp. 334–344.
6. Python Software Foundation, “Tkinter — Python interface to Tcl/Tk,” Python 3 Documentation, 2024. [Online]. Available: https://docs.python.org/3/library/tkinter.html
7. R. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, “Usage, costs, and benefits of continuous integration in open-source projects,” in Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE), Singapore, 2016, pp. 426–437.