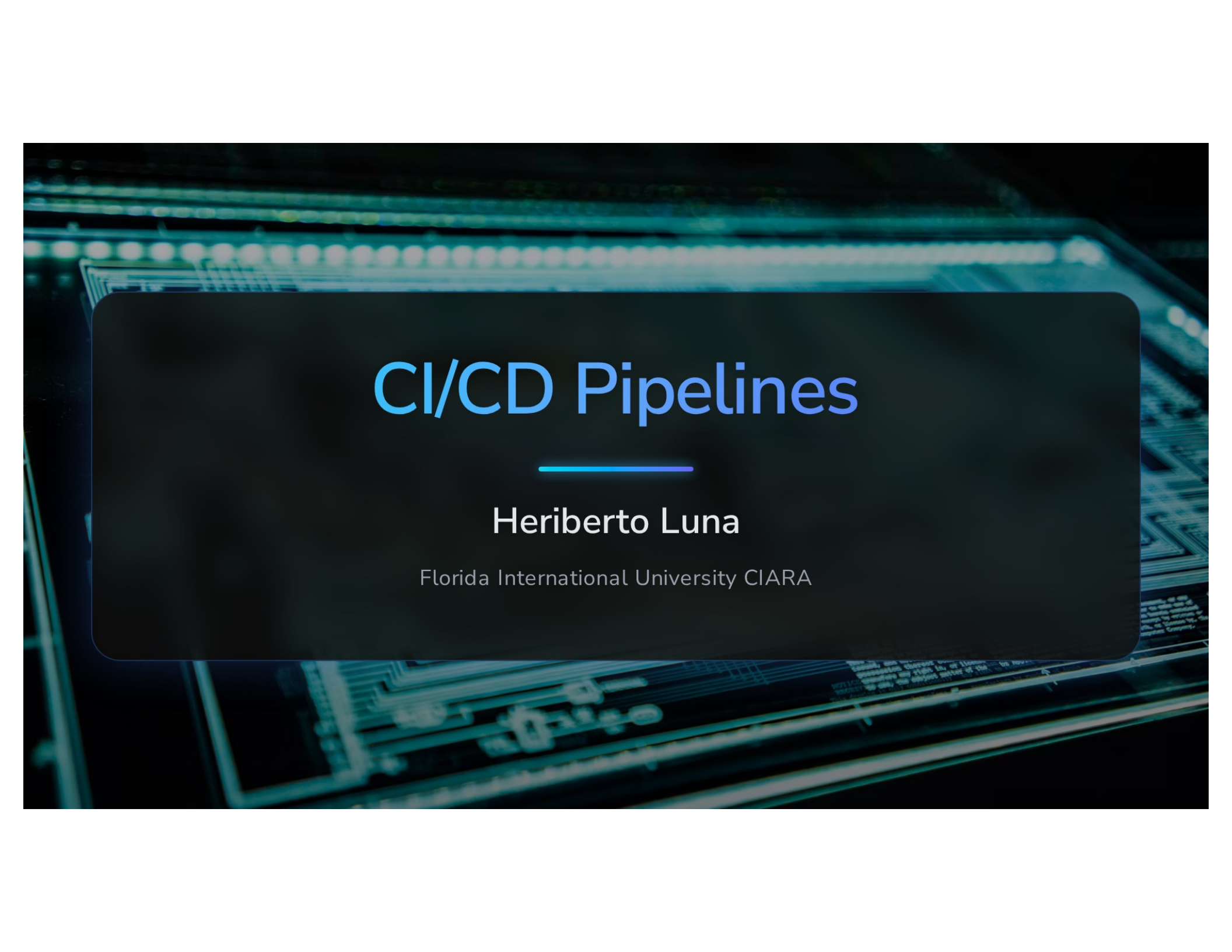




CI/CD Pipelines

Heriberto Luna

Florida International University CIARA

Table of Contents

Core topics and presentation roadmap



01. What is CI/CD?

Fundamentals, pipelines & continuous security



02. The Research Problem

Modern pipeline challenges & research questions



03. Business Value & Benefits

Why stakeholders care about automation ROI



04. Local & External Automation

From Git hooks to cloud-based runners



05. CI & CD Implementations

React frontend vs. advanced Rust/Python backend



06. GitOps Architecture

Orchestration with k3s, ArgoCD & Ansible



07. AI Integration & Improvements

Automated reviews with CodiumAI PR Agent

What is CI/CD?



Continuous Integration

The practice of automating the different workflows that need to take place in order to integrate/add code into a project.

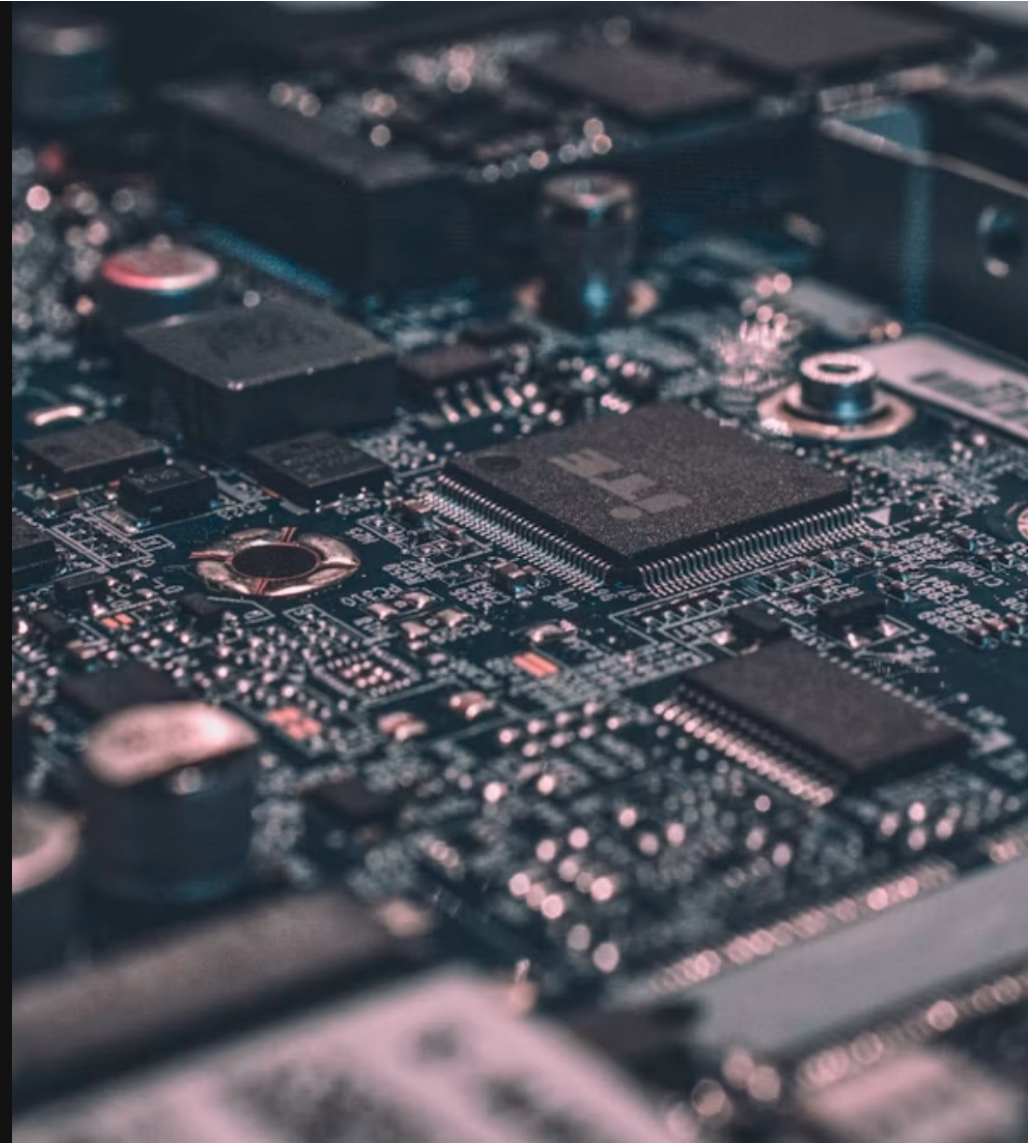


Continuous Deployment

The practice of automating the releases of your software into a production environment. Whenever you finish writing code and adding it into your project, the continuous deployment pipeline is responsible for automatically releasing it into production.

What are CI/CD Pipelines?

Your CI/CD pipeline is your set of automated workflows in order to continuously integrate code into a project and release it into production.



Continuous Security (CS)

Continuous Security (DevSecOps)



SAST & Secret Auditing

Static Application Security Testing scans uncompiled source code, Dockerfiles, and IaC for vulnerability patterns, SQL injection risks, and leaked API credentials.

✓ Static Code & IaC Analysis



DAST & Runtime Defense

Dynamic Application Security Testing checks for security vulnerabilities while your code is running.

✓ Dynamic Exploit Scanning



CI-CS-CD Paradigm: Standard CI validates functional logic; Continuous Security verifies structural integrity and prevents supply-chain exploits before production deployment.

The Research Problem

Addressing the challenges of modern web application delivery

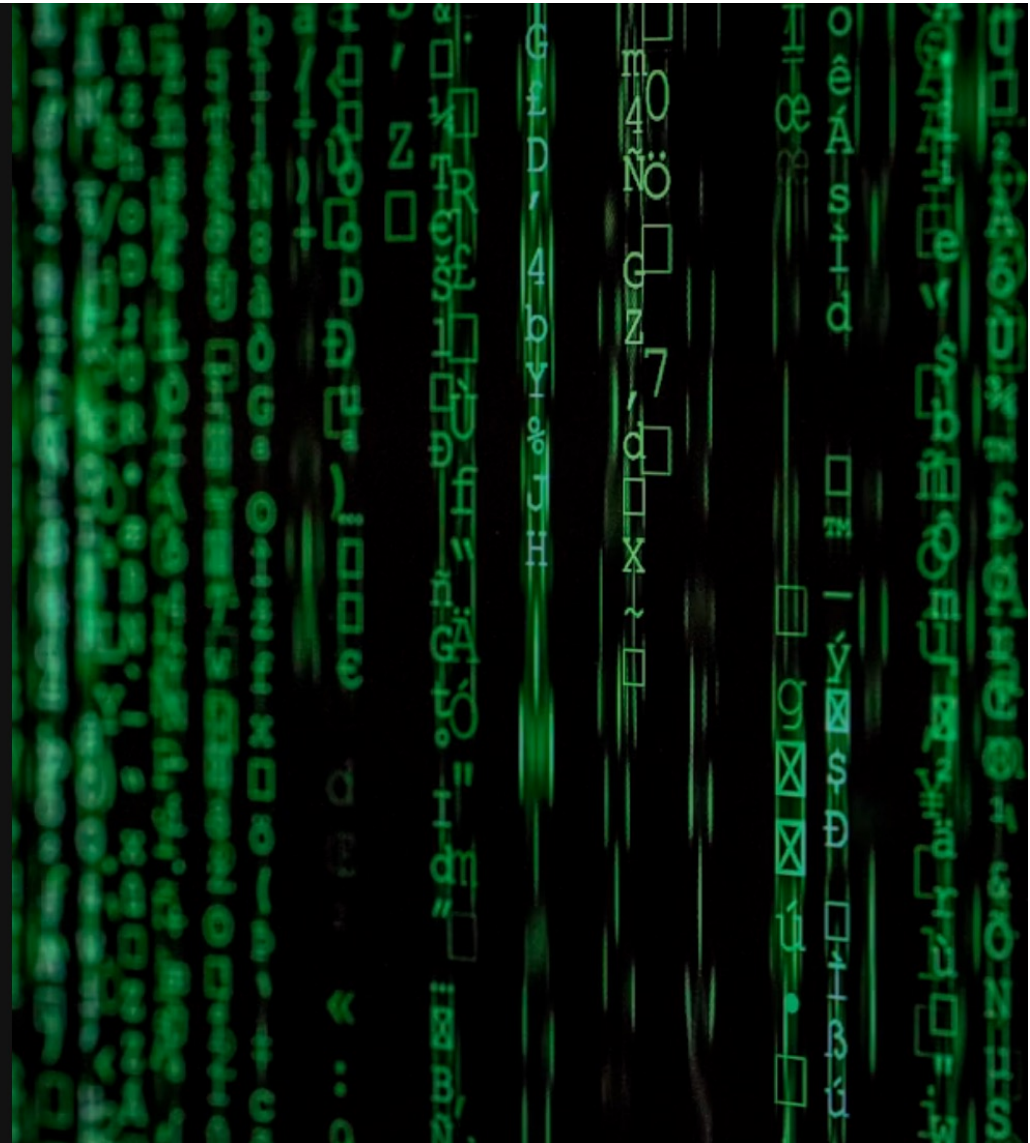
As modern software systems grow in complexity, organizations struggle to standardize workflows across multiple languages, package managers, and build systems. This research focuses on solving the following challenges:

❓ RESEARCH QUESTION 1

"How can a modern CI/CS/CD pipeline for frontend and backend web applications be developed?"

🤖 RESEARCH QUESTION 2

"How can AI be integrated into a modern CI/CS/CD pipeline?"



Business Value & ROI

Why non-technical stakeholders care about CI/CD automation

⚠ The Manual Delivery Trap

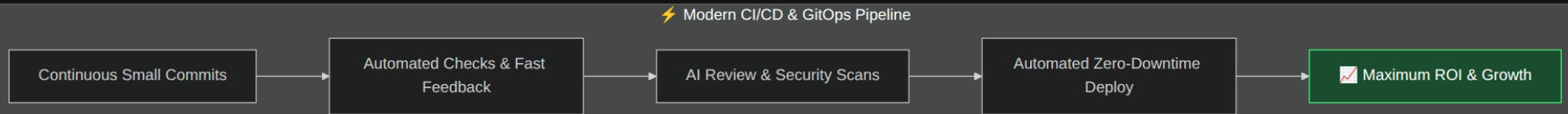
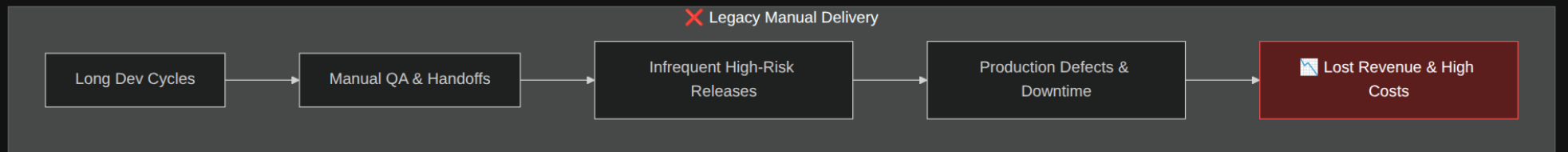
- **Slow Time-to-Market:** Manual handoffs and testing delay critical feature releases by weeks or months.
- **High Production Risk:** Human error during manual deployments leads to costly outages and emergency hotfixes.
- **Resource Waste:** Engineering hours are consumed by repetitive manual verification instead of building core business value.

📈 The Automated CI/CD Advantage

- **Rapid Innovation:** Continuous integration enables releasing small, verified updates multiple times a day.
- **Predictable Quality:** Automated testing and AI reviews catch regressions before they ever impact customers.
- **Maximized ROI:** Drastically reduces operational overhead, lowering total cost of ownership (TCO) and boosting customer retention.

Visualizing Business Impact

Comparing pipeline efficiency and financial outcomes



Story Introduction



You are a software developer, and you want to create a website.

You learn about a front-end framework like **Vue.js** and start writing code.

It's hard to keep track of changes, so you use **Git** to manage the version history of the application.

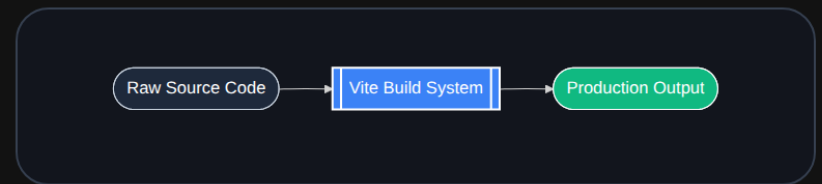
You want to work with others and safely store code, so you utilize a **GitHub repository**.

The Problem & The Build System

🤔 How will you manage your project?

How will you build your code into something usable?

You need a build system; let's say you use Vite in order to build your Vue.js front-end web application.



Static Analysis: The Need

You want to keep your code high quality and avoid syntax errors, so you introduce **static analysis tools** like a linter and a formatter. Static analysis tools scan your raw unexecuted code for issues.

✗ Before Formatting

```
const add=(a,b )⇒ {  
  return a+b;}  
}
```

✓ After Formatting

```
const add = (a, b) ⇒ {  
  return a + b;  
};  
}
```

Linters and Formatters Explained

≡ Formatter

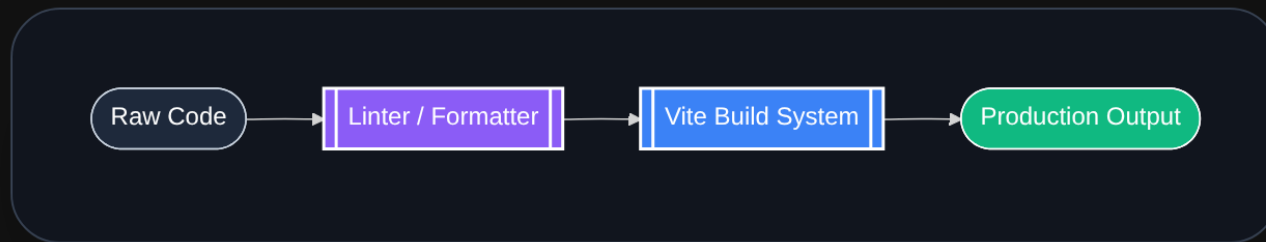
A formatter is very simple; it checks and corrects the style of your code. It makes sure it follows proper formats and is easy to read. It makes sure you use proper indentation, quote usage, semicolon usage, etc.

⚠ Linter

A linter will check your code for **syntax errors**; it will make sure that it is logically sound, it will make sure it follows best practices, and it will introduce no “anti-patterns” or things that would make your code worse.

Updating the Pipeline (Static Analysis)

Let's add Static Analysis to our workflow.



```
import * as serviceWorker from './serviceWorker';
ReactDOM.render((
  <BrowserRouter>
    <Switch>
      <Route path="/login" component={LoginPage} />
      <ProtectedRoute exact={true} component={HomePage} />
      <ProtectedRoute path="/settings" component={SettingsPage} />
      <ProtectedRoute component={Dashboard} />
    </Switch>
  </BrowserRouter>
), document.getElementById('root'));

// If you want to use a service worker, you can uncomment the following code
serviceWorker.unregister();
```

OUTPUT DEBUG CONSOLE TERMINAL

Testing?

Your code is now beautiful, properly formatted, and linted. It can also build and be displayed, but how are you going to test it, making sure that every single component works properly as it should?

You could test it manually by yourself; you could have a list of things that you need to click on after every change, but that is too tedious and will not scale.

What happens if you have a thousand buttons that you have to click on? That is when you utilize a testing framework. You basically just write a program that will test your application for you, and you can reuse it whenever you want.

Types of Tests

There are many types of tests that you can do, but the two main types of tests that you will be utilizing are **unit** and **end-to-end (E2E)** tests.

Unit Tests

Unit tests are written in order to make sure that small parts of your code work like individual functions or components.

End-to-End (E2E) Tests

E2E tests make sure your entire application works. They test the application end-to-end, covering multiple sections of it at once. It represents common workflows that your program will perform (e.g., submitting a form, storing data in a DB).

Testing in Action

Unit Test (Vitest)

```
import { expect, test } from 'vitest'
import { add } from './math'

test('adds 1 + 2 to equal 3', () => {
  expect(add(1, 2)).toBe(3)
})
```

💡 **Fast Verification:** Unit tests execute in milliseconds in Node.js, verifying isolated business logic before bundling.

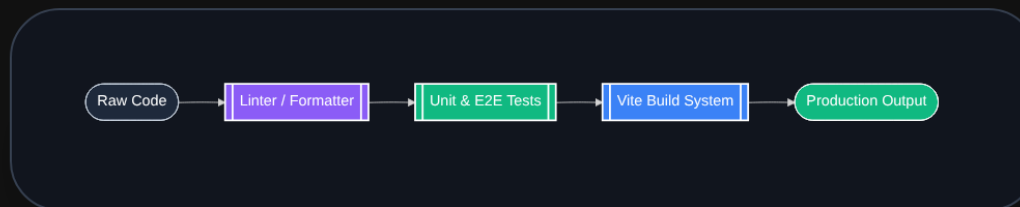
E2E Test (Cypress)

```
describe('Form Submission', () => {
  it('submits successfully', () => {
    cy.visit('/form')
    cy.get('input[name="email"]').type('user@test.com')
    cy.get('button[type="submit"]').click()
    cy.contains('Success!')
  })
})
```

💡 **User Journey Testing:** E2E tests run in a real browser, verifying complete user workflows and DOM interactions.

Updating the Pipeline (Testing)

Now our pipeline has automated testing.



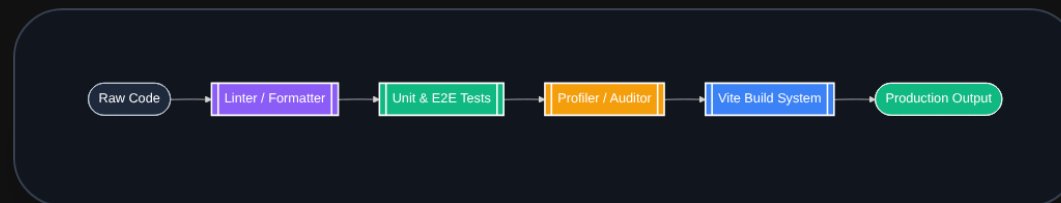
Profiling and Auditing

Now that your code is properly tested, you need to make sure that it is performant and memory efficient. This is when you use a profiler or auditor.



Updating the Pipeline (Profiling)

The final step of our local code validation.



Local Automation: The Need



The Workflow Problem

You have all of the tools and you probably have your own workflow, but you have to do it all **manually**. How can you run it locally and automatically?



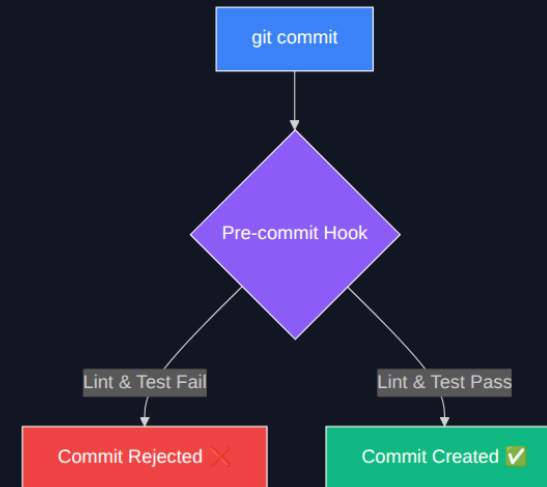
The Solution: Git Hooks

The most common way is to utilize **Git hooks and scripts**. You can have some sort of script to download all of your dependencies and set up the git hooks.

How Git Hooks Work

Whenever you perform an action with Git, usually like committing code or pushing code to a repository, you can run a custom script.

So whenever you try to commit code, you can make sure that your linters, formatters, and unit tests run so that everything functions properly before the commit is saved.



External Automation

⚠ The Trust Problem

You know how to validate your code locally, but what about externally within your repository? That is the most important part; someone could disable these automated workflows locally and push bad code to the repo.

💻 The CI Server Solution

Your repo should validate everything as well. For GitHub, you could use something like GitHub Actions to execute your continuous integration pipeline in the same exact way, in a clean environment.



Additional Functionality

You have the basics of your CI pipeline done. There are a few more things that you could add:



Automating changelog generation



Automating project management &
Scrum



Automating releases



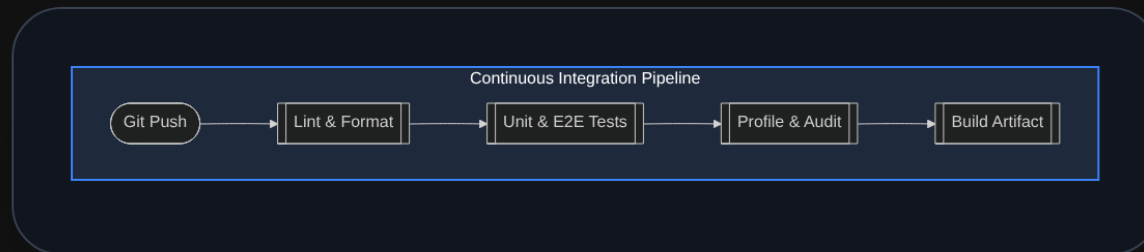
Automating application building



Testing across multiple
environments

Continuous Integration Complete

Your continuous integration pipeline is now done, but how are you going to utilize it in multiple different projects and handle modifications? This is when a scaffolding tool like `cookiecutter` or `copier` comes in, which allows you to create reusable templates that are easy to manage.



The Next Frontier

We've built a pristine artifact.
But an artifact sitting in a registry provides **zero value** to the user.

How do we ship it safely?

The Challenge of Continuous Deployment

With the continuous integration pipeline now complete, we can now go into the **continuous deployment pipeline**.

Continuous deployment is **not** easy, because there are many different types of projects and applications that can be built and a lot of ways to release them.

Libraries & Binaries

You could have a library or an executable; you compile it for many different systems and architectures and then plop it onto a website for others to install.

Web Applications

It can get more complicated; you could have a website that you want to serve, and you may then need a database, a reverse proxy like NGINX, and a backend REST API.

Initial Continuous Deployment & IaC

To be very modern, we could use Kubernetes, ArgoCD, and GitOps practices.

Let's go on a tangent; to deploy an application, you need an environment where it will run. To set up this environment you could configure it by hand and type out all of the commands, or you could use infrastructure as code (IaC).

You can declaratively define all of your infrastructure code. Then you can take it one step further; you can define not just infrastructure, but everything as code ("X as code"). You can have policies, infrastructure, configuration, security, and much more as code.

deployment.yaml

```
# Example: Kubernetes Deployment (IaC)
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
        - name: web
          image: my-web-app:latest
```

Enter GitOps

All of this code can be version controlled by Git, and you can use Git to move forward or back through the history of all of your "X as code" files.

This is useful because if something breaks in a staging environment or prod, you can quickly go back to a previous version that worked.

The GitOps Philosophy

Then you can take it one step further with "GitOps". In GitOps, infrastructure as code is treated as normal code, and all processes like reviews and testing are applied to it.

The Dedicated Repository

We can have a dedicated repository, most likely on GitHub, which will have all of our X-as-code files, and they can then be run through a continuous integration pipeline where they can be linted and tested.

Secure Execution in GitOps

Pull Requests as Gates

They can also only be changed via reviewed pull requests, which makes everything more secure since these files define our infrastructure and environments; we have to make sure that every change is deliberate and functional.

CD Application

We can then have a continuous deployment pipeline, which will apply all of these x-as-code files to their respective environments.



How to implement CD?

How will continuous deployment work? A pull model will be used.

Push Model (Danger)

We could have a sort of central control tower with access to every single server, which would push and manage the changes, but this is dangerous.

Pull Model (Secure)

Instead, every system can be told where the GitOps repository is and to "follow it" and pull changes from it.

This can be done via polling or, the more efficient method, hooks. This is very straightforward for Kubernetes.

ArgoCD

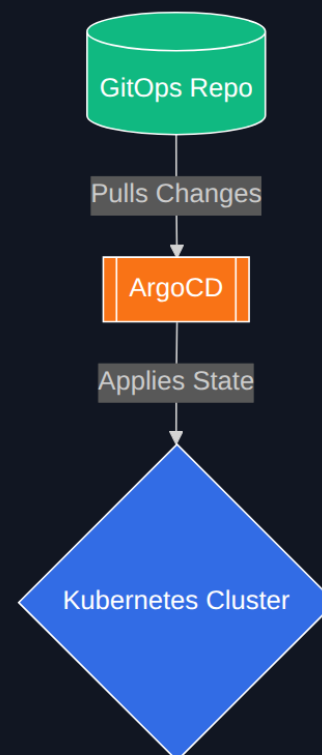


The Kubernetes GitOps Controller

Kubernetes has a tool called Argo CD, which, when installed, we can just point to the GitOps repository, and it will constantly pull new changes and apply them to the cluster.

It also makes sure that the cluster strictly follows the manifest files within the GitOps repo; the GitOps repo is the only source of truth, and you cannot make changes locally. If someone makes changes to the cluster locally, then ArgoCD will revert them.

If this is too strict, it can be changed within the configuration of ArgoCD, but what about the rest of the system? How do we initially install Kubernetes and set everything up?



Ansible & Infrastructure

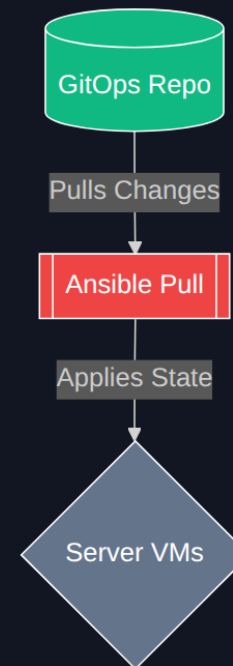


Automating the Bare Metal

This is where Ansible then comes in. Ansible can initially configure the servers, but then Ansible uses a push model. This is unsecure, so after the initial configuration we need to remove the access that Ansible has to the servers, but then how do we constantly apply new changes?

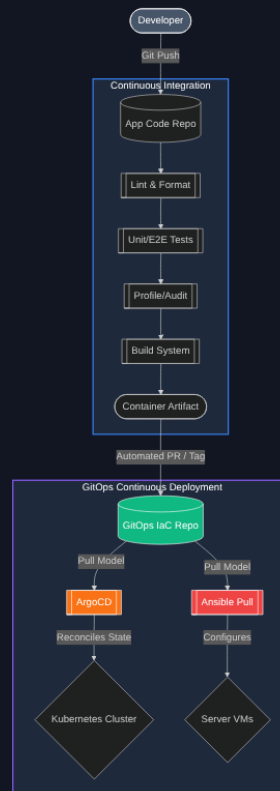
Ansible has a module called "ansible pull", which can work just like ArgoCD, but it is not as robust.

To make Ansible work just like ArgoCD, I have a list of tools and workflows in mind that I can test, but only after all of this. We then finally have a single or multiple GitOps repositories, which are the only source of truth, and manage all the servers and clusters.



Final CI/CD Pipeline

This is our final, complete CI/CD and GitOps Pipeline architecture.



Dual CI Pipelines Overview

Tailoring continuous integration to specialized project ecosystems

React Frontend Pipeline

Designed for modern UI applications and JavaScript/TypeScript ecosystems.

npm

Vite / Vitest

ESLint / Prettier

Storybook

Playwright

Unlighthouse

Advanced Backend Pipeline

More advanced pipeline engineered for high-performance Rust & Python services.

PreK

Ruff / Pyright

Pytest + Coverage

Scalene

Makefile / Bash

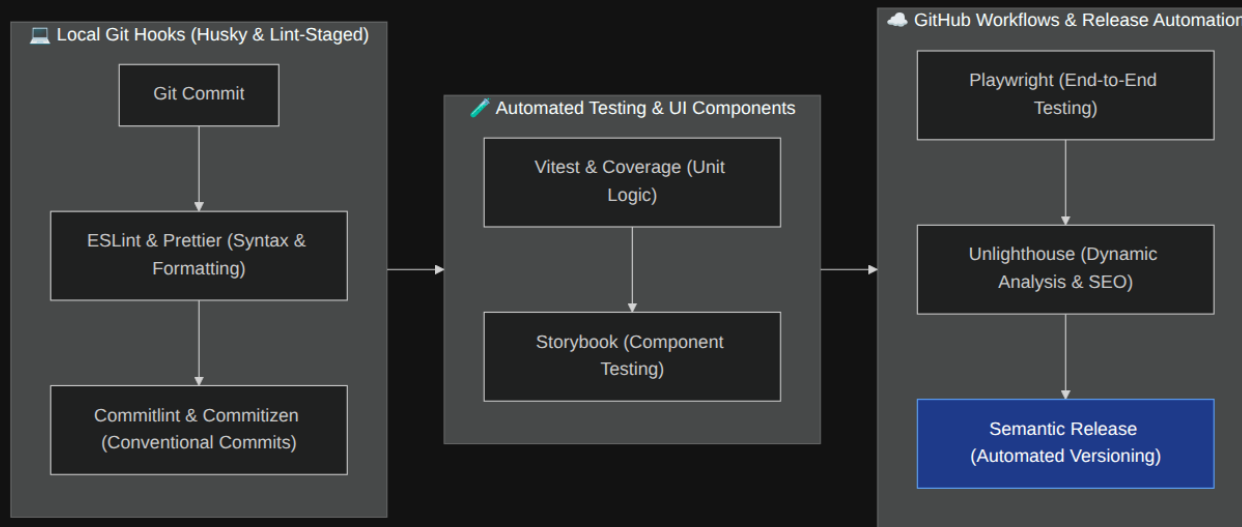
Release Please



Why do they differ? Different projects use different languages, package managers, and build systems, and some of the steps of a continuous integration pipeline rely on these choices. Different package managers each have their own automation tools available, and different languages need their own static analysis tools, like linters.

React Frontend CI Pipeline

Architecture and tooling for modern user interfaces



✓ Local Verification

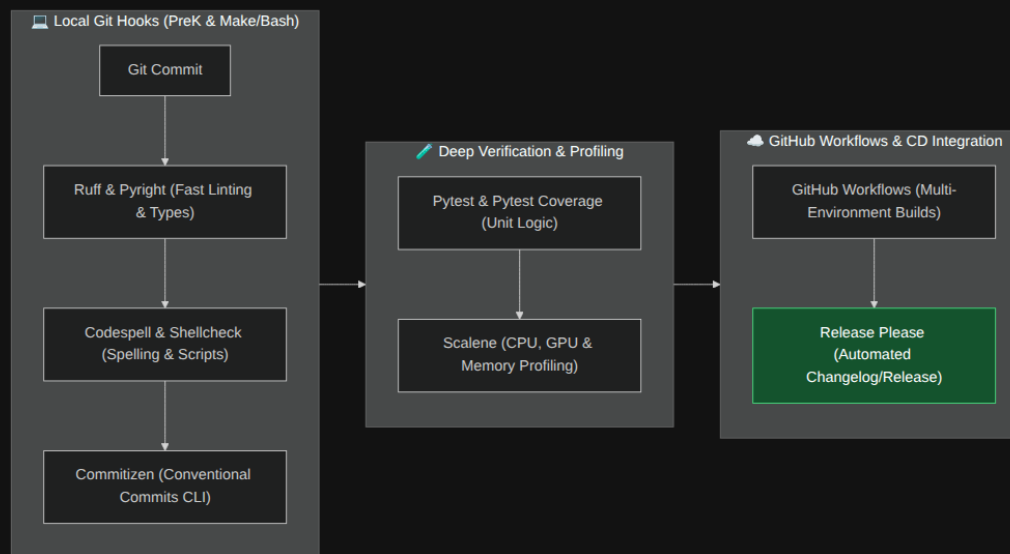
Utilizes Husky and lint-staged to run quick syntax checks (ESLint, Prettier) and enforce commit message standards (Commitlint) prior to local git commit.

☁ Cloud Execution

GitHub Workflows execute heavy suites like Playwright E2E tests, profile dynamic runtime metrics with Unlighthouse, and trigger Semantic Release.

Advanced Backend CI Pipeline

High-performance Rust & Python static and dynamic verification



⚡ Why More Advanced?

Utilizes PreK (a high-speed Rust rewrite of pre-commit) along with Makefile and Bash scripts to orchestrate complex multi-language checks and type verification (Pyright).

📊 Dynamic Profiling

Integrates Scalene for deep resource profiling and utilizes Release Please to automate GitHub release workflows and version management.

Local CI Automation & The Boy Scout Rule

Ensuring code quality at the commit boundary without slowing down developers

Git-Triggered Execution

The two pipelines follow the same structure: committing changes locally utilizing Git triggers the local part of the CI pipeline. Husky and PreK ensure distinct scripts run during Git actions.

</> Fast Static Analysis

Husky and PreK run scripts to perform quick checks on the code, executing linters and formatters—in this case, Ruff, ESLint, and Prettier.

🔗 The Boy Scout Rule

Tools like PreK and lint-staged make sure that **only the changes that you are about to commit** are verified and tested.

- Drastically reduces execution time during local commits.
- Prevents large existing repositories with legacy code from causing the pipeline to constantly fail.

"Always leave the codebase cleaner than you found it—without getting bogged down by legacy debt."

Automated Testing & Local Verification

Balancing test rigor with local development velocity

Tiered Test Execution

Automated tests verify code functionality using unit testing frameworks (pytest, Vitest) and component tools (Storybook).

 *End-to-end tests (Playwright) are time-consuming, so they do not run at this local stage.*

Coverage Diff

Coverage reports check test percentages against strict failure thresholds. Using `coverage diff` generates reports *only for code about to be committed*, adhering to the Boy Scout rule.

Standardizing with Commitizen

After all checks and tests pass, Commitizen provides an interactive command-line interface to author commits.

```
? Select the type of change you are committing:
> feat: A new feature
fix: A bug fix
docs: Documentation only changes
```

Enforces the Conventional Commits specification for clean logs, documentation, and automated downstream changelog generation.

External CI Pipeline in the Cloud

Rerunning validation and dynamic profiling across GitHub Workflows

GitHub Workflows

Pushing code and opening a pull request triggers cloud workflows—selected for their simplicity and native GitHub integration. Reruns all local checks plus heavy end-to-end suites (Playwright).

Dynamic Analysis

Builds and tests code across multiple runtime environments. Utilizes dynamic analysis tools to profile software as it runs: Scalene (resource profiling) and Unlighthouse (live site quality & SEO).

Semantic Releases

After AI code review, tools like release-please and semantic-release automate version bumps and package distributions following the Semantic Versioning specification.



Continuous Deployment & GitOps

Orchestrating containers with Kubernetes, ArgoCD, and Ansible

Ansible IT Automation

Ansible automates server management, downloading k3s to provision a lightweight Kubernetes cluster across dedicated development, staging, and production server nodes.

Container Orchestration

Kubernetes manages large amounts of containerized applications— isolating software packages so they can be easily transported, scaled, and executed anywhere.

ArgoCD & "X as Code"

Ansible installs and configures ArgoCD, a GitOps continuous deployment controller for Kubernetes.

- Constantly pulls X as code (infrastructure and configuration files defining environment state) from a side GitOps GitHub repository.
- Fetches the final build images from registries and applies configurations to the cluster.
- Ensures the entire project is deployed, synchronized, and ready.

GitOps Validation & CIARA Feedback

Ensuring infrastructure reliability and real-world developer alignment



Independent GitOps CI

The side GitOps repository containing all infrastructure as code was independently linted and verified using its own dedicated CI pipeline to prevent cluster misconfigurations.

✓ IaC Verified & Linted



Creation Documentation

After finalizing the pipeline architectures, comprehensive creation documentation and architectural runbooks were written to ensure seamless onboarding and reproducible deployments.

✓ Runbooks Authored



CIARA Developer Feedback

The complete pipeline was distributed to software developers across major CIARA research projects to gather practical feedback and validate real-world performance.

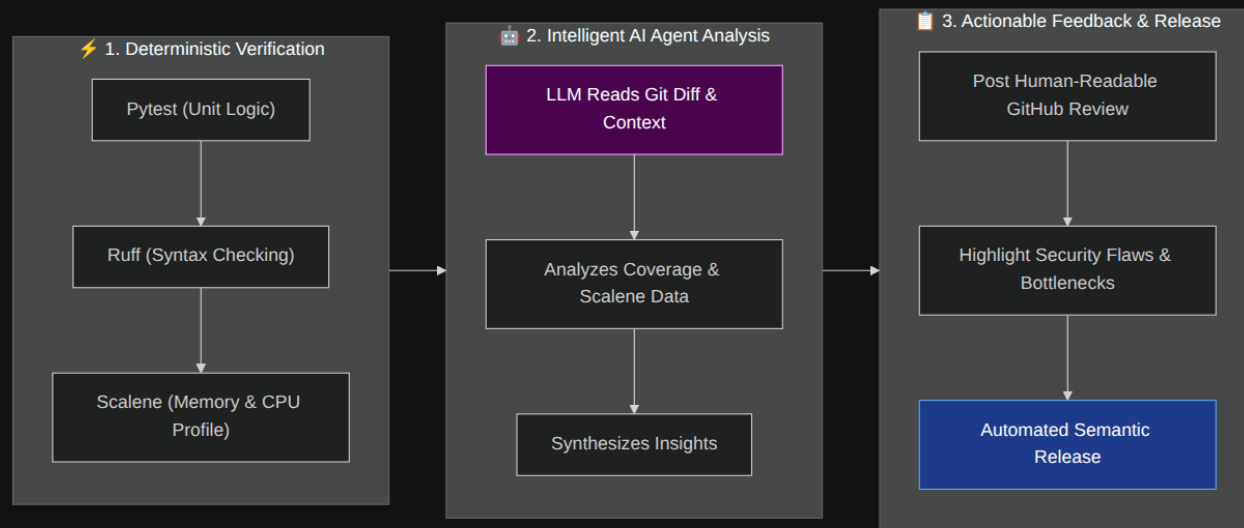
✓ AW-SDX | Kytos-NG | Amlight



Outcome: Bridged the gap between academic architecture and production-grade network engineering workflows.

AI Integration: CodiumAI PR Agent

Bridging deterministic verification with intelligent LLM code analysis



1. Deterministic First

Standard tools run first within GitHub Workflows: Pytest tests logic, Ruff checks syntax, and Scalene profiles memory usage.

2. Contextual AI Review

The LLM reads git diffs, checks surrounding files for deep context, and correlates test coverage with Scalene profiling data.

3. Automated Release

After human-readable review and feedback, automated tools (release-please, semantic-release) finalize semantic versioning.

AI Reviewer Capabilities & Future Roadmap

Enhancing developer velocity with intelligent feedback and continuous evolution

What the AI Agent Delivers

- **Logical Inconsistencies:** Points out subtle logical flaws in commit diffs by evaluating surrounding file context.
- **Advanced Security Scans:** Highlights vulnerability patterns and security risks missed by standard static analysis scans.
- **Performance Refactoring:** Suggests concrete code refactoring directly based on Scalene memory and CPU bottlenecks.
- **Seamless Release:** Paves the way for automated releases via release-please and semantic-release.

Possible AI Improvements

- **Autonomous Remediation:** Evolving from review suggestions to auto-generating pull requests that fix identified Scalene bottlenecks and security flaws.
- **Domain-Specific Tuning:** Fine-tuning the LLM reviewer on historical codebases and networking architectures specific to CIARA projects (AW-SDX, Kytos-NG).
- **Dynamic Test Generation:** Automatically writing unit test assertions for untested code paths uncovered during coverage diff checks.

"This was a lot, but it is the CI pipeline in full."

Conclusions



Automation ensures code quality, testing, and formatting scale seamlessly.



CI pipelines catch errors before they ever reach production.



IaC and "X as Code" allow us to version-control our entire infrastructure.



A Pull-based GitOps model (ArgoCD / Ansible Pull) ensures peak security and a single source of truth.

Many Thanks To



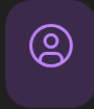
CIARA



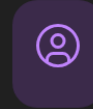
The McNair Program



Dr. Vasilka Chergarova



Vinicius Arcanjo



Jeronimo Bezerra

Resources & Next Steps

Thank you for attending. If you want to dive deeper into the code, architecture diagrams, or connect with me, please scan the QR code.

 github.com/HeriLFIU

 linkedin.com/in/heribertoluna



SCAN FOR SLIDES & CODE

Questions?

Thank you for your time and attention.

Heriberto Luna

FLORIDA INTERNATIONAL UNIVERSITY CIARA