

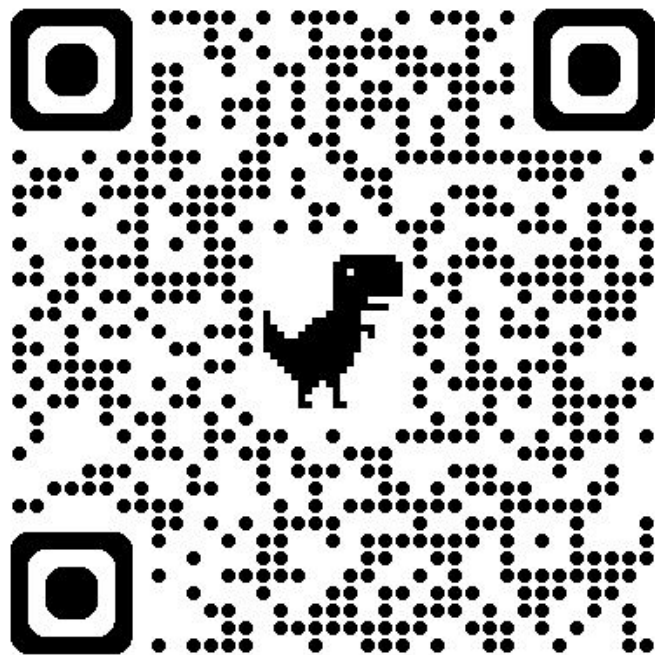


Team 20

Connor Allen: Project Manager

Isaac Hutchinson: UI Designer & Lead Architect

Lon Danna: Software Developer & Test Engineer



GitHub: [github.com/Connor-Allen10/codemon](https://github.com/Connor-Allen10/codemon)

# The\_Problem\_and\_Motivation

## The Problem

- Debugging\_Anxiety
- Cognitive\_Load
- Bugs = Failure\_State

## Our Solution

- Gamified\_Intuition
- Build\_Confidence
- Pattern\_Recognition

**Target: Deep C++ Competency**

# Technical\_Approach

## Bug\_Tiering

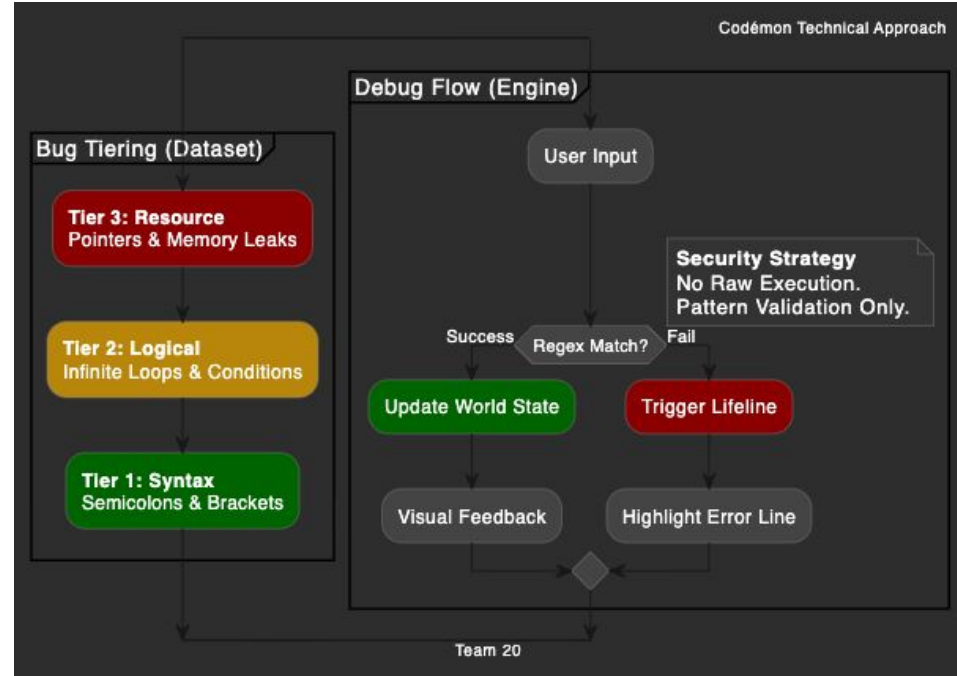
Syntax (Low), Logic (Med),  
Resource/Memory (High).

## Interaction:

Code Editor UI overlay with  
"Compiler Feedback" lifelines.

## Validation:

Regex-based Pattern Matching Engine  
(No raw execution).



# Software Architecture – Components & Responsibilities

## Presentation Layer (SFML)

- Renders game world and UI
- Captures player code input
- Displays validation feedback

## Domain Layer (DebugEngine)

- Orchestrates gameplay logic
- Routes code input to validators
- Translates validation results into game state updates

## Validation Layer (Validators)

- Applies regex-based bug detection
- Categorizes bugs by tier (syntax / logic / memory)
- Returns structured ValidationResult

## Infrastructure Layer (JSONLoader)

- Loads bug definitions, species data, and progression rules
- Enables data-driven content without recompiling

# Data Organization & Component Interfaces

## Data Storage (JSON – Data-Driven Design)

**File:** /assets/data/bugs.json

Each bug entry includes:

- id
- tier (syntax / logic / memory)
- pattern (regex)
- hint\_text
- failure\_message
- success\_effect

## Component Interface Contract

**SFML → DebugEngine**

- Sends: std::string userCode
- Receives: GameStateUpdate

**DebugEngine → Validator**

- Sends: std::string userCode
- Receives: ValidationResult
  - isValid
  - bugType
  - feedbackMessage

## Example:

```
{
  "id":
  "missing_semicol
on",
  "tier":
  "syntax",
  "pattern":
  ";$",
  "hint": "Check
statement
termination"
}
```

# Key Architecture Decisions & Alternatives

## Decision 1 – Game State Management

**Chosen:** Centralized State Stack (Play / Battle / Menu / Pause)

**Alternative:** Independent game objects with distributed state

### Why chosen

- Predictable transitions + easy pause/menu overlays
- Isolates state logic → simpler debugging + unit tests

### Tradeoffs

- Less flexible for highly dynamic worlds
- Requires strict stack discipline to avoid state bugs

## Decision 2 – Bug Validation Strategy

**Chosen:** Regex-based pattern validation (no execution)

**Alternative:** AST parsing (Clang) or executing student code in a sandbox

### Why chosen

- Safe (no untrusted code execution)
- Deterministic feedback supports learning goals
- Lightweight + cross-platform for term project scope

### Tradeoffs

- Limited semantic understanding vs AST
- Some complex bugs must be approximated by patterns

# Risk Assessment & Mitigation Plan

## SFML Build & Linking

- Likelihood: **Med** | Impact: **High**
- Mitigation: CMake toolchain + early CI builds

## Validator Logic Complexity

- Likelihood: **High** | Impact: **Med**
- Mitigation: Tiered difficulty + unit tests per rule

## State Transition Bugs

- Likelihood: **Med** | Impact: **Med**
- Mitigation: Centralized state stack + assertions

## Scope Creep (Content Growth)

- Likelihood: **Med** | Impact: **Med**
- Mitigation: Lock Tier 1-2 features early

## Usability vs Learning Goals

- Likelihood: **Low** | Impact: **Med**
- Mitigation: Early playtesting + iterative feedback

# Testing Plan and Process

## **Tool:**

GitHub Issues for bug tracking.

## **Strategy:**

- Automated unit tests for Validator classes; Manual usability tests for UI.
- Black box testing to focus on user input and observable outcomes
- White box testing to test all internal logic: Debugging logic, error handling

# Schedule\_and\_Conclusion

**February:** Core Engine Complete.

**March:** Zone 1 & UI Polish.

**March:** Final Rigorous Testing.