# Cloud Deployment DSL Compiler

**Team G-Force**

**Indian Institute of Information Technology, Design and Manufacturing, Kancheepuram**

**Course: Compiler Design | Guide: Prof. Jagadeesh**

# Project Overview

**The Cloud Deployment DSL Compiler is a domain-specific language compiler designed to parse, analyze, and generate deployment plans for cloud infrastructure. It simplifies specifying providers, regions, services, resource requirements, and optimization goals.**

# Motivation & Objective

- Automate cloud deployment configurations through a simple DSL.
- Provide a compiler pipeline that transforms DSL scripts into optimized JSON plans.
- Demonstrate compiler design principles (Lexical, Syntax, Semantic, Optimization, Target Code ).

# Compiler Architecture

**The compiler consists of the following stages:**

**1) Lexical Analysis – Tokenises DSL input (lexer.l)**

**2) Syntax Analysis – Validates grammar and builds AST (parser.y)**

**3) Semantic Analysis – Constructs logical model using AST (ast.c/.h)**

**4) Optimisation – Refines deployment based on constraints (optimiser.c)**

**5) Code Generation – Outputs optimised JSON plan (codegen.c)**

**6) Execution Control – Handles file I/O and compiler flow (main.c)**

# File: lexer.l (Lexical Analyser)

- Uses Flex to scan and convert input text into tokens.
- Recognises keywords such as DEPLOYMENT, PROVIDER, REGION, SERVICE, etc.
- Handles identifiers, numbers, and string literals.
- Passes tokens to Yacc parser with semantic values.

# Files: ast.c / ast.h (AST Management)

- Defines C structures for Deployment, Service, Constraint, and Environment.
- Provides helper functions to create, connect, and print AST nodes.
- Manages dynamic memory allocation and relationships between deployment components.

# File: parser.y (Syntax Analyser)

- Defines grammar for the Cloud DSL using Yacc.
- Parses tokens into structured Abstract Syntax Tree (AST).
- Manages hierarchical constructs like DEPLOYMENT, SERVICE, and OPTIMIZE blocks.
- Calls AST construction functions defined in ast.c.

# File: optimiser.c (Optimisation Phase)

- Analyses constraints (cost, latency) and adjusts deployment parameters.
- Example: increases service replicas to reduce latency while staying within cost limits.
- Demonstrates compiler optimisation phase.

# File: codegen.c (Code Generation)

- Generates JSON deployment plan from optimised AST.
- Converts internal data structures into structured JSON.
- Handles formatting and ensures readable, valid output.

# File: main.c (Compiler Driver)

- Acts as the entry point of the compiler.
- Opens input .ddl file and invokes lexer, parser, optimiser, and code generator.
- Outputs final deployment plan to terminal or JSON file.

# Example Input and Output

**Input DSL (sample_env.ddl):**

```
DEPLOYMENT MyApp {
 PROVIDER aws;
 REGION us-east-1;
 SERVICE web {
  IMAGE "nginx:latest";
  CPU 2;
  MEM 1024;
  REPLICAS 2;
 };
 OPTIMIZE { COST 500; LATENCY 80; };
}
```

**Output JSON:**

```
{
 "deployment": "MyApp", "provider": "aws", "region": "us-east-1", ...
}
```

# Team Contributions

- **Vijay Pranav (CS23B1073)**
  **Grammar design, parser, AST .**

- **ManiKanta (CS23B1072)**
  **Token definitions, debugging, testing.**

- **Avinash Kumar (CS23B1080)**
  **Lexical analyser and optimisation logic.**

- **Nisarg (CS23B1090)**
  **Code generation , integration and final report preparation.**

# Results & Future Scope

**Results:**

- **Successfully parses DSL and generates JSON plans.**
- **Implements core compiler phases.**

**Future Enhancements:**

- ***Add more keywords (SECURITY, STORAGE, NETWORK).***
- **Provide GUI front-end for DSL input.**
- **Enable real cloud API integration (AWS, Azure).**

## High-level architecture and connectivity Explanation

The compiler follows a classical pipeline:-
***Lexical analysis (Flex)** — lexer.l reads the input .ddl file and converts text into tokens (keywords, identifiers, numbers, strings, symbols).
***Parsing (Yacc/Bison)** — parser.y consumes tokens and applies grammar rules to recognize program structure (deployment, services, env blocks, constraints). Semantic actions in the parser build the AST using functions in ast.c.
***AST (Semantic model)** — ast.h / ast.c define the data structures (Deployment, Service, EnvVar, Constraint) and helper functions to construct and manipulate them.
***Optimization** — optimizer.c takes the AST and applies optimization passes (example: increase replica counts to satisfy constraints).
***Code generation** — codegen.c serializes the optimized AST into a JSON-formatted deployment plan.
***Main driver** — main.c ties everything together: opens input, calls parser, runs optimizer, calls emitter, cleans up.
At each stage data flows via C data structures (pointers to structs), so components are linked by function calls and shared headers.

# Conclusion

**This project demonstrates how compiler design concepts can be applied to real-world automation. It transforms a high-level deployment description language into a structured, optimised plan, bridging the gap between human-readable scripts and machine-deployable configurations.**