# Complier Design

## Project Report

### (Cloud Deployment DSL Compiler )

### Computer Science Department

### November 3, 2025

# Contents

**5. Supported Features**



**6. Challenges and Solutions**



**7. Results and Future Enhancements**



8. **Debugging And Testing**



**9. Conclusion**

# By Team G-force

# Members


**M1: R U Vijay Pranav (CS23B1073)**

**M2: ManiKanta (CS23B1072)**

**M3: Avinash Kumar (CS23B1080)**

**M4: Nisarg (CS23B1090)**

# 1. Cloud Deployment DSL Compiler — Detailed Project Explanation

## 1. 1 Project overview (Abstract)

The Cloud Deployment DSL Compiler is a compact domain-specific compiler that parses human-friendly deployment descriptions (a small DSL), constructs an internal Abstract Syntax Tree (AST), performs simple optimizations, and emits a machine-friendly JSON deployment plan.

The front end is implemented with a Flex lexer (lexer.l) and a Yacc/Bison parser (parser.y), the AST and semantic helpers are in ast.h / ast.c, optimizations live in optimizer.c, and the final plan is produced by codegen.c.

A Makefile automates building Process.

### *1.2 Motivation*

In today's rapidly evolving cloud computing ecosystem, developers and system administrators frequently face the challenge of **configuring and deploying applications** across multiple cloud providers such as AWS, Azure, and Google Cloud. Each provider has its own configuration syntax, deployment format, and optimization strategies, making multi-cloud deployments **error-prone, time-consuming, and difficult to maintain**.

To address these challenges, we were motivated to design a **Domain Specific Language (DSL)** that simplifies and standardizes the way cloud deployments are defined.
Our DSL allows users to specify key deployment parameters—such as provider, region, services, resources, and optimization goals—in a clean, human-readable format.

The **compiler** for this DSL then parses, analyzes, and converts these specifications into a structured deployment plan (in JSON format) that can be easily extended for automation or integration with cloud APIs.

In short, our motivation lies in:

- Reducing the complexity of writing cloud deployment configurations.
- Creating a unified syntax for cloud resource descriptions.
- Automating the generation of deployment plans using compiler design principles.
- Demonstrating how traditional compiler techniques (lexing, parsing, code generation) can be applied to **real-world DevOps scenarios**.

---

### *1.3 Objectives*

The key objectives of the *Cloud Deployment DSL Compiler* project are:

1. **Design a custom DSL** for describing cloud deployments in a simplified, human-readable form.
2. **Implement lexical and syntax analysis** using Flex and Yacc (Lex and Yacc tools) to parse the DSL code.
3. **Construct an Abstract Syntax Tree (AST)** to represent deployment structures such as services, constraints, and environments.
4. **Apply basic optimization techniques** to adjust parameters (e.g., CPU, replicas) for cost and latency improvements.
5. **Generate structured output (in JSON format)** to represent deployment plans that can be further processed or used for cloud automation.
6. **Ensure modular design**, with separate files for lexical analysis, parsing, AST construction, optimization, and code generation.
7. **Demonstrate compiler workflow**, including scanning, parsing, semantic representation, optimization, and output generation.
8. **Provide an extensible foundation**, allowing future enhancements like multi-cloud deployment translation or integration with real APIs.

# 2. High-level architecture and pipeline Explanation

### 2.1 Complier pipeline overview

The compiler follows a classical pipeline:

1. **Lexical analysis (Flex)** — lexer.l reads the input .ddl file and converts text into tokens (keywords, identifiers, numbers, strings, symbols).
2. **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.
3. **AST (Semantic model)** — ast.h / ast.c define the data structures (Deployment, Service, EnvVar, Constraint) and helper functions to construct and manipulate them.
4. **Optimization** — optimizer.c takes the AST and applies optimization passes (example: increase replica counts to satisfy constraints).
5. **Code generation** — codegen.c serializes the optimized AST into a JSON-formatted deployment plan.
6. **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.

## 2.4 Project Structure:

```
Cloud Deployment DSL Compiler/
|
├── lexer.l
├── parser.y
├── ast.h
├── ast.c
├── optimizer.h
├── optimizer.c
├── codegen.h
├── codegen.c
├── main.c
├── Makefile
├── sample_env.ddl
└── plan.json
```

# 3. How to build and run

Assuming you are on Ubuntu / WSL with flex, bison (or yacc) and a C toolchain installed:

```
# Clean and build

make clean

make

# Run with sample input and capture JSON output

./cloudopt sample_env.ddl > plan.json


# Or debug build:

cc -g y.tab.c lex.yy.c ast.c optimizer.c codegen.c main.c -o cloudopt_dbg -ll

./cloudopt_dbg sample_env.ddl
```

Extra Notes Point :

- If you want debug logs separated from JSON, direct debug prints to stderr and JSON to stdout.
- If you see "Clock skew detected" warnings, run make -B to force rebuild or fix host/WSL time sync.

# 4. File-by-file explanation

Below I Have explain each important file in the project, what it contains, and how it connects to other files.

## 4.1 lexer.l — lexical analyzer (Flex)

**Purpose:** Tokenize the DSL input. Recognizes keywords, identifiers, numbers, strings, and punctuation.

**Key parts:**

**Header block:**

```
%{
#include "y.tab.h"
#include <string.h>
#include <strings.h>
#include <stdlib.h>
%}
%option caseless

```

**%option caseless**

- Includes y.tab.h (parser-generated token definitions).
- %option caseless (make matching case-insensitive). Additionally, keyword checks use strcasecmp for robustness.

- **Token rules:**
    - Whitespace ignored:

```
[ \t\r\n]+ ;
"{" { return LBRACE; }
"}" { return RBRACE; }
";" { return SEMI; }
"," { return COMMA; }
"=" { return EQUALS; }   /* important for ENV assignments */
```

**String literal rule:**

- \"[^\"]*\" { yylval.str = strdup(yytext + 1); yylval.str[strlen(yylval.str)-1] = '\0'; return STRING; }

```
\"[^\"]*\" { yylval.str = strdup(yytext + 1); yylval.str[strlen(yylval.str
    )-1] = '\0'; return STRING; }
```

**Number rule:**

- [0-9]+ { yylval.num = atoi(yytext); return NUMBER; }

```
[0-9]+ { yylval.num = atoi(yytext); return NUMBER; }
```

**Identifier / keyword rule:**

- [a-zA-Z_][a-zA-Z0-9_-]* {
-     if (strcasecmp(yytext,"deployment")==0) return DEPLOYMENT;
-     ...
-     yylval.str = strdup(yytext);
-     return ID;
- }

```
[a-zA-Z_][a-zA-Z0-9_-]* {
    if (strcasecmp(yytext,"deployment")==0) return DEPLOYMENT;
    ...
    yylval.str = strdup(yytext);
    return ID;
```

This block checks keywords (case-insensitive) and otherwise returns ID with semantic value in yylval.str.

**Connections:**

- Returns tokens declared in parser.y (via y.tab.h).
- Populates yylval so parser actions can receive lexeme strings/numbers.

**Important implementation notes:**

- strcasecmp matches uppercase/lowercase keywords.
- Returning EQUALS for = is crucial because parser grammar expects a named token EQUALS.

## 4.2 parser.y — grammar & semantic actions (Yacc/Bison)

**Purpose:** Parse sequences of tokens into AST nodes and wire them together; also handles some basic semantic checks.

**Top-level structure:**

%union lists types stored in yylval:

%union {

char *str;

int num;

Service *svc;

Constraint *con;

Deployment *dep;

EnvVar *env;

}

```
%union {
    char *str;
    int num;
    Service *svc;
    Constraint *con;
    Deployment *dep;
    EnvVar *env;
}

```

============================================

**token declares tokens and types:**

%token <str> ID STRING

%token <num> NUMBER

%token DEPLOYMENT PROVIDER REGION SERVICE IMAGE CPU MEM REPLICAS OPTIMIZE COST LATENCY ENV TRUE FALSE

%token LBRACE RBRACE SEMI COLON LPAREN RPAREN COMMA

%token EQUALS

```
%token <str> ID STRING
%token <num> NUMBER
%token DEPLOYMENT PROVIDER REGION SERVICE IMAGE CPU MEM REPLICAS OPTIMIZE
    COST LATENCY ENV TRUE FALSE
%token LBRACE RBRACE SEMI COLON LPAREN RPAREN COMMA
%token EQUALS


```

==============================================

**deployment grammar (core):**

deployment:

DEPLOYMENT ID LBRACE

{ root_deployment = create_deployment($2); free($2); }

dep_body RBRACE

{ $$ = root_deployment; root_deployment = NULL; };

```

deployment:
    DEPLOYMENT ID LBRACE
        { root_deployment = create_deployment($2); free($2); }
    dep_body RBRACE
        { $$ = root_deployment; root_deployment = NULL; }
;


```

This creates the Deployment AST node early (so dep_item actions append details to it).

- dep_item includes PROVIDER, REGION, service, and OPTIMIZE blocks.
- The OPTIMIZE rule consumes the trailing SEMI to match the DSL style: OPTIMIZE { ... };.

==============================================

# service rule:

```
service:
  SERVICE ID LBRACE svc_body RBRACE SEMI
  {
    Service *s = create_service($2);
    free($2);
    add_service(root_deployment, s);
  };
```

```

service:
    SERVICE ID LBRACE svc_body RBRACE SEMI
    {
        Service *s = create_service($2);
        free($2);
        add_service(root_deployment, s);
    }
;

```

Each service is created and added to the current deployment.

- svc_item supports IMAGE STRING;, CPU NUMBER;, MEM NUMBER;, REPLICAS NUMBER;, and ENV { ... };.
- env_list / env_item:

==============================================

env_item:

ID EQUALS STRING SEMI { $$ = create_env($1,$3); free($1); free($3); }

;

```


env_item:
    ID EQUALS STRING SEMI { $$ = create_env($1,$3); free($1); free($3); }
;


```

==============================================

create_env returns an EnvVar* which is appended and attached to the latest service by add_env.

# Constraint parsing:

constraint:

COST NUMBER SEMI { $$ = create_constraint(CON_COST,$2); }

| LATENCY NUMBER SEMI { $$ = create_constraint(CON_LATENCY,$2); };

```

constraint:
    COST NUMBER SEMI { $$ = create_constraint(CON_COST,$2); }
  | LATENCY NUMBER SEMI { $$ = create_constraint(CON_LATENCY,$2); }
;


```

**Semantic actions:**

- The parser calls functions in ast.c to create nodes and link them (e.g., create_deployment, create_service, set_provider, add_service, create_constraint, append_constraint, create_env, append_env, add_env).

**Error handling:**

- yyerror(const char *s) prints parse errors to stderr. During development, breakpointing yyerror in gdb was used for debugging.

**Connections:**

- Relies on tokens from lexer.l and AST helpers from ast.h/c.
- After successful parse, root_deployment points to a filled Deployment AST passed to optimizer and codegen.

## 4.3 ast.h — AST type definitions and API

**Purpose:** Declares the in-memory structures used as the semantic model and prototypes for manipulation functions.

**Key types:**

```
typedef struct EnvVar {
    char *key;
    char *value;
    struct EnvVar *next;
} EnvVar;

typedef struct Service {
    char *name;
    char *image;
    int cpu;
    int mem;
    int replicas;
    EnvVar *env;
    struct Service *next;
} Service;

typedef struct Constraint {
    ConType type; // CON_COST or CON_LATENCY
    int value;
    struct Constraint *next;
} Constraint;

typedef struct Deployment {
```

char *name;

char *provider;

char *region;

Service *services;

Constraint *constraints;

} Deployment;

```
typedef struct EnvVar {
    char *key;
    char *value;
    struct EnvVar *next;
} EnvVar;

typedef struct Service {
    char *name;
    char *image;
    int cpu;
    int mem;
    int replicas;
    EnvVar *env;
    struct Service *next;
} Service;

typedef struct Constraint {
    ConType type; // CON_COST or CON_LATENCY
    int value;
    struct Constraint *next;
} Constraint;

typedef struct Deployment {
    char *name;
    char *provider;
    char *region;
    Service *services;
    Constraint *constraints;
} Deployment;
```

**Key functions declared:**

- Creation and mutation: create_deployment, create_service, add_service, latest_service, set_service_*, create_constraint, append_constraint, set_constraints.
- Environment helpers: create_env, append_env, add_env.
- Utilities: free_deployment, print_deployment.

**Connections:**

- Used by parser semantic actions and by optimizer & codegen.

## 4.4 ast.c — AST implementation

**Purpose:** Implements functions declared in ast.h: allocation, linking, string duplication, append functions, and cleanup.

**Important details:**

- Memory allocation: strdup used for string fields to isolate lexeme memory.
- create_service initializes env = NULL and default values (e.g., cpu=1, mem=512, replicas=1).
- append_env and append_constraint walk linked lists to append new nodes.
- free_deployment walks entire structure freeing all memory (services, env lists, constraints) — prevents leaks.
- print_deployment produces a human-readable debug printout used during testing.

**Connections:**

- Called by parser.y to build AST, by optimizer to update AST, and by codegen to emit output. free_deployment is called at program end or on parse failure.

—

## 4.5 optimizer.c / optimizer.h — optimization pass

**Purpose:** Run simple optimization passes on the AST — this demonstrates compiler transformation and constraint handling.

**Example functionality (typical):**

- Increase service replicas if constraints or some heuristic demands it (e.g., if latency is high, increase replicas).
- Constraint enforcement or selection of configuration values to meet COST or LATENCY constraints.

**Implementation notes:**

- The optimizer takes Deployment* and may produce a modified Deployment* (in place).

- The optimizer prints a debug message showing the AST before and after optimization (for demo).
- This module is intentionally simple to keep project scope reasonable while demonstrating a transformation stage.

**Connections:**

- Called from main.c after parsing and before code generation.

—

## 4.6 codegen.c — code / JSON generation

**Purpose:** Serialize the optimized AST into a JSON-like deployment plan that external tools can consume.

**Key features:**

- emit_service(Service *s, int first) prints JSON for a single service (name, image, cpu, mem, replicas, env array).
- emit_plan(Deployment *d) prints the full JSON object (deployment, provider, region, services array, constraints array).

**Formatting notes:**

- The emitted JSON is plain printf-based; careful comma handling and first flags avoid trailing commas.
- Env vars emitted either as empty array [] or list of key/value pairs like {"KEY": "VALUE"} depending on implementation.

**Connections:**

- Called by main.c after optimizer, using Deployment*.

—

## 4.7 main.c — program driver

**Purpose:** CLI entrypoint, orchestrates build pipeline:

Typical steps:

1. Open input file (or read stdin).
2. Set yyin to that file so flex reads it.
3. Call yyparse() to run parsing and AST construction.
4. If parse succeeds: print debug AST (optional), call optimizer, print optimized AST (optional), call emit_plan to write JSON.
5. Cleanup: free AST, close files.

**Command-line usage:**

Usage: ./cloudopt <input.dsl>

```
Usage: ./cloudopt <input.dsl>
```

**Connections:**

- Uses yyparse, emit_plan, optimize functions. It is the glue.

## 4.8 Makefile

**Purpose:** Automates code generation and compilation:

Typical commands:

- yacc -d parser.y or bison -d parser.y → produces y.tab.c and y.tab.h.
- flex lexer.l → produces lex.yy.c.
- cc compile: cc -O2 -Wall y.tab.c lex.yy.c ast.c optimizer.c codegen.c main.c -o cloudopt -ll.

Targets:

- all or default: build cloudopt.
- clean: remove generated files (y.tab.c, y.tab.h, lex.yy.c, cloudopt, *.o).

**Notes:**

- If you use GNU bison, %code requires is accepted; POSIX yacc warns but still works. If warnings are unwanted, call bison directly or remove %code requires and include ast.h in parser header blocks.

## 4.9 sample_env.ddl and sample_input.ddl

**Purpose:** Example DSL input files used for testing.

**Example content:**

```
DEPLOYMENT MyApp {
  PROVIDER aws;
  REGION us-east-1;

  SERVICE web {
    IMAGE "nginx:latest";
    CPU 1;
    MEM 512;
    REPLICAS 1;
    ENV {
      PORT = "8080";
      MODE = "production";
    };
  };

  OPTIMIZE {
    COST 500;
```

```
        LATENCY 80;
    };
}
```

```
DEPLOYMENT MyApp {
    PROVIDER aws;
    REGION us-east-1;

    SERVICE web {
        IMAGE "nginx:latest";
        CPU 1;
        MEM 512;
        REPLICAS 1;
        ENV {
            PORT = "8080";
            MODE = "production";
        };
    };

    OPTIMIZE {
        COST 500;
        LATENCY 80;
    };
}
```

These files demonstrate:

- Keyword use (DEPLOYMENT, SERVICE, IMAGE, etc.)

- ENV block with KEY = "VALUE"; syntax
- OPTIMIZE block with constraints

**Connections:**

- Passed to ./cloudopt for parsing and to produce plan.json.

—

# 5. Data flow and runtime example

**Parsing and AST build**

- Lexer tokenizes: DEPLOYMENT → DEPLOYMENT token; MyApp → ID with yylval.str = "MyApp".
- Parser deployment rule executes semantic action: root_deployment = create_deployment("MyApp").
- Subsequent dep_item rules call set_provider, set_region, and for each service, create_service and add_service. ENV rules create EnvVar objects appended to Service->env.

**Optimization**

- optimizer.c inspects root_deployment->constraints and root_deployment->services, adjusts replicas, or tunes resource allocations.

**Code generation**

- codegen.c traverses the final Deployment AST and prints JSON. Example: a service with replicas changed from 1 to 2 by optimizer is reflected in output.

**Sample run**

./cloudopt sample_env.ddl > plan.json

cat plan.json

This writes JSON plan to plan.json. Debug and progress text printed to stderr remains on console; if you redirect both stdout/stderr to file you will capture everything.

```
./cloudopt sample_env.ddl > plan.json
cat plan.json

```

—

## 6. Debugging & testing notes

- Use gdb to debug parser and find exact syntax error points:
  - gdb ./cloudopt_dbg
  - break yyerror
  - run sample_env.ddl
  - bt to view parser call stack and inspect yytext, yylval, and root_deployment.
- Flex/Bison mismatch symptoms:
  - If keywords return as ID, check lexer's keyword comparisons; use strcasecmp or %option caseless.
  - If = is not recognized, ensure lexer has "=" { return EQUALS; }.
- Use make clean && make -B to force full rebuilds when "clock skew" warnings occur.

## 7. Design choices & rationale

- **Flex + Bison**: classical compiler front-end tools; educational and robust.
- **AST in plain C structs**: simple, memory-efficient, and easy to manage in C — demonstrates pointer management, allocation, and cleanup.
- **JSON output**: practical target format (can be consumed by other tools).
- **Small optimization pass**: demonstrates that compiler can *transform* code — a key compiler capability (not just parsing).

- **Case-insensitive keywords**: practical for user convenience

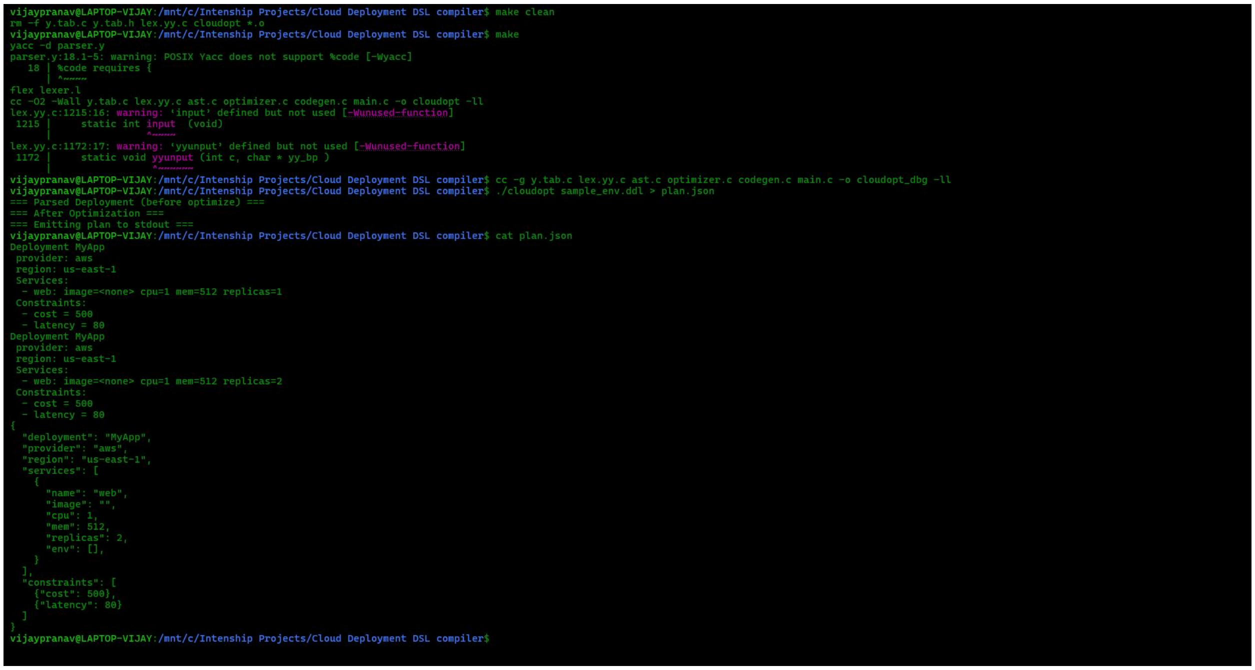
# 8. Limitations and future extensions

This implementation is intentionally small for a student project. Possible extensions to make the project richer:

- Add **more DSL constructs**: storage, network, autoscale rules, health checks.
- Add **type checking and semantic error messages** (e.g., resource limits, conflicting constraints).
- Implement a **cost model** and pick concrete provider options based on constraints.
- Replace printf JSON generation with a proper JSON library for correctness and quoting.
- Implement **unit tests** for lexer/parser using bison --debug or regression tests.
- Build a **GUI** or web interface to write DSL and view generated plan.
- Add support for **multiple deployments** in one file.

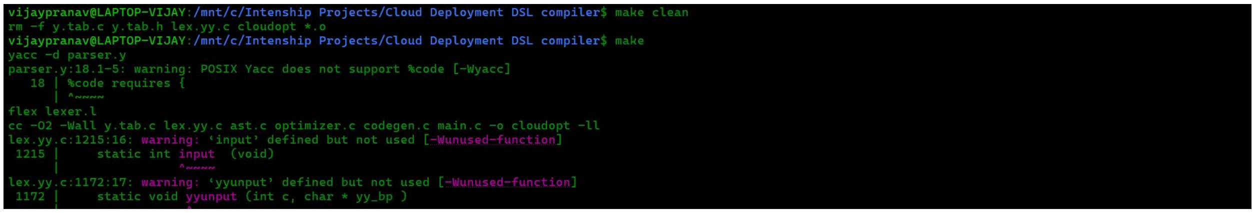
# Conclusions Obtained:

## Total Output Obtained



```
vijaypranav@LAPTOP-VIJAY:/mnt/c/Intenship Projects/Cloud Deployment DSL compiler$ make clean
rm -f y.tab.c y.tab.h lex.yy.c cloudopt *.o
vijaypranav@LAPTOP-VIJAY:/mnt/c/Intenship Projects/Cloud Deployment DSL compiler$ make
yacc -d parser.y
parser.y:18.1-5: warning: POSIX Yacc does not support %code [-Wyacc]
   18 | %code requires {
      | ^~~~~
flex lexer.l
cc -O2 -Wall y.tab.c lex.yy.c ast.c optimizer.c codegen.c main.c -o cloudopt -ll
lex.yy.c:1215:16: warning: ‘input’ defined but not used [-Wunused-function]
 1215 |     static int input  (void)
      |                ^~~~~
lex.yy.c:1172:17: warning: ‘yyunput’ defined but not used [-Wunused-function]
 1172 |     static void yyunput (int c, char * yy_bp )
      |                 ^~~~~~~
vijaypranav@LAPTOP-VIJAY:/mnt/c/Intenship Projects/Cloud Deployment DSL compiler$ cc -g y.tab.c lex.yy.c ast.c optimizer.c codegen.c main.c -o cloudopt_dbg -ll
vijaypranav@LAPTOP-VIJAY:/mnt/c/Intenship Projects/Cloud Deployment DSL compiler$ ./cloudopt sample_env.ddl > plan.json
=== Parsed Deployment (before optimize) ===
=== After Optimization ===
=== Emitting plan to stdout ===
vijaypranav@LAPTOP-VIJAY:/mnt/c/Intenship Projects/Cloud Deployment DSL compiler$ cat plan.json
Deployment MyApp
 provider: aws
 region: us-east-1
 Services:
  - web: image=<none> cpu=1 mem=512 replicas=1
 Constraints:
  - cost = 500
  - latency = 80
Deployment MyApp
 provider: aws
 region: us-east-1
 Services:
  - web: image=<none> cpu=1 mem=512 replicas=2
 Constraints:
  - cost = 500
  - latency = 80
{
  "deployment": "MyApp",
  "provider": "aws",
  "region": "us-east-1",
  "services": [
    {
      "name": "web",
      "image": "",
      "cpu": 1,
      "mem": 512,
      "replicas": 2,
      "env": [],
    }
  ],
  "constraints": [
    {"cost": 500},
    {"latency": 80}
  ]
}
vijaypranav@LAPTOP-VIJAY:/mnt/c/Intenship Projects/Cloud Deployment DSL compiler$
```

## Enviroment Creation Using Make file



```
vijaypranav@LAPTOP-VIJAY:/mnt/c/Intenship Projects/Cloud Deployment DSL compiler$ make clean
rm -f y.tab.c y.tab.h lex.yy.c cloudopt *.o
vijaypranav@LAPTOP-VIJAY:/mnt/c/Intenship Projects/Cloud Deployment DSL compiler$ make
yacc -d parser.y
parser.y:18.1-5: warning: POSIX Yacc does not support %code [-Wyacc]
   18 | %code requires {
      | ^~~~~
flex lexer.l
cc -O2 -Wall y.tab.c lex.yy.c ast.c optimizer.c codegen.c main.c -o cloudopt -ll
lex.yy.c:1215:16: warning: ‘input’ defined but not used [-Wunused-function]
 1215 |     static int input  (void)
      |                ^~~~~
lex.yy.c:1172:17: warning: ‘yyunput’ defined but not used [-Wunused-function]
 1172 |     static void yyunput (int c, char * yy_bp )
```

## Compilation

```
vijaypranav@LAPTOP-VIJAY:/mnt/c/Intenship Projects/Cloud Deployment DSL compiler$ cc -g y.tab.c lex.yy.c ast.c optimizer.c codegen.c main.c -o cloudopt_dbg -ll
vijaypranav@LAPTOP-VIJAY:/mnt/c/Intenship Projects/Cloud Deployment DSL compiler$ ./cloudopt sample_env.ddl > plan.json
=== Parsed Deployment (before optimize) ===
=== After Optimization ===
=== Emitting plan to stdout ===
vijaypranav@LAPTOP-VIJAY:/mnt/c/Intenship Projects/Cloud Deployment DSL compiler$ cat plan.json
```

## Jason File Obtained As Output

```
=== Emitting plan to stdout ===
vijaypranav@LAPTOP-VIJAY:/mnt/c/Intenship Projects/Cloud Deployment DSL compiler$ cat plan.json
Deployment MyApp
 provider: aws
 region: us-east-1
 Services:
  - web: image=<none> cpu=1 mem=512 replicas=1
 Constraints:
  - cost = 500
  - latency = 80
Deployment MyApp
 provider: aws
 region: us-east-1
 Services:
  - web: image=<none> cpu=1 mem=512 replicas=2
 Constraints:
  - cost = 500
  - latency = 80
{
  "deployment": "MyApp",
  "provider": "aws",
  "region": "us-east-1",
  "services": [
    {
      "name": "web",
      "image": "",
      "cpu": 1,
      "mem": 512,
      "replicas": 2,
      "env": [],
    }
  ],
  "constraints": [
    {"cost": 500},
    {"latency": 80}
  ]
}
vijaypranav@LAPTOP-VIJAY:/mnt/c/Intenship Projects/Cloud Deployment DSL compiler$
```