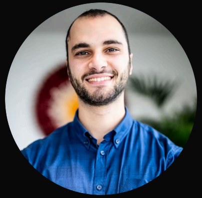
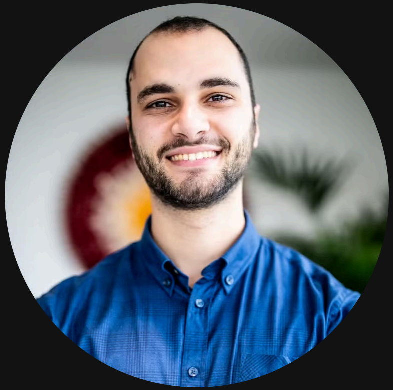
From Scripts to Strategy

The Past, Present, and Future of Infrastructure as Code



Amr Awad

Staff Software Engineer @ Cur8 Capital



Amr Awad

Linkedin

amrawad.com

hello@amrawad.com

Adplist

Currently

Staff Software Engineer @ Cur8 Capital

Building a sharia compliant investment platform

Founder @ Lunchflow

Built a SaaS app connecting UK & EU banks to Lunch Money using Open Banking.

Previously

Founding Engineer @ Tempo (YC W15)

First full-time employee. Helped scale the company to 250+ people and raise $300M+ in funding.

Top Rated Freelancer @ Upwork

Working across multiple domains, including: machine learning, computer vision, cloud architecture and DevOps.

Embedded Software Engineer @ Valeo

Developed software for autonomous driving systems and advanced driver assistance features.

Agenda

- What is Infrastructure as Code (IaC)?
- The Evolution of IaC
- Next Generation of IaC tooling
- Making the Right Choice

What is Infrastructure as Code?

Infrastructure as Code

Your Application

- Your core business logic

Everything Else

- Compute Resources
- Storage Solutions
- Networking
- Monitoring & Alerting

Infrastructure as Code

Automated
No manual clicking in consoles

Version Controlled
Track changes, rollback when needed

Reproducible
Same code = Same infrastructure

Testable
Validate before applying

Collaborative
Review infrastructure changes like code

```
# AWS Infrastructure Example
resource "aws_instance" "web" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"

  tags = {
    Name = "WebServer"
    Environment = "Production"
  }
}

resource "aws_s3_bucket" "logs" {
  bucket = "my-app-logs"
}
```

Example Terraform Script

Infrastructure as Code Benefits

Consistency

Eliminate configuration drift

Speed

Rapid infrastructure deployment

Scale

Manage complex infrastructure

Collaboration

Team-wide infrastructure visibility

The Evolution of Infrastructure as Code

1976

Make

```
install:
    mkdir -p ~/myapp
    cp config.json ~/myapp/
    chmod +x ~/myapp/start.sh

configure:
    echo "PORT=3000" > ~/myapp/.env
    echo "DEBUG=true" >> ~/myapp/.env
    ./start.sh
```

- Configuration Management
- Utilizes shell commands
- Focuses on build processes

Trade-offs

- ✅ Simple syntax
- ✅ Runs on any Unix system.
- ❌ No state tracking
- ❌ Single machine

1993

</> CFEngine

```
bundle agent example {
  files:
    "/etc/nginx/nginx.conf"
      perms => mog("644", "root", "root"),
      create => "true",
      edit_line => nginx_config;

  processes:
    "nginx"
      restart_class => "restart_nginx";
}
```

- Built by Mark Burgess while pursuing a postdoctorate in theoritical physics
- Enables managing multiple Unix workstations
- Introduced a declarative DSL to manage complexity

Trade-offs

- ✅ Multiple machines with a lightweight agent
- ❌ Steep learning curve due to complex syntax
- ❌ Limited adoption

2000-2006

The Scale Problem

Traditional Approach

- Buy physical servers
- Set up in data centers
- Long procurement cycles
- High upfront costs

Enter AWS EC2 (2006)

- Virtual machines on demand
- Pay-as-you-go model
- Minutes vs. months to deploy
- Democratized scaling

Growing Demands

- Internet boom
- Web-scale applications
- Need for rapid scaling

Infrastructure scaling became everyone's challenge, not just large enterprises ...

2005

Puppet

```
package { 'nginx':
  ensure => installed
}

service { 'nginx':
  ensure  => running,
  enable  => true,
  require => Package['nginx']
}
```

- Declarative DSL
- State Management

Trade-offs

- Idempotent Operations
- Configuration Drift Detection
- Much simpler than CF Engine

2009

Chef

```ruby
package 'nginx' do
  action :install
end

template '/etc/nginx/nginx.conf' do
  source 'nginx.conf.erb'
  owner 'root'
  group 'root'
  mode '0644'
  notifies :restart, 'service[nginx]'
end

service 'nginx' do
  action [:enable, :start]
  supports status: true, restart: true, reload: true
end
```

- Full Ruby DSL
- Procedural approach to configuration
- Developer-friendly workflow

Trade-offs

- ✅ Powerful Ruby programming model
- ✅ Rich ecosystem of cookbooks
- ❌ Steeper learning curve for non-developers

2012

Ansible

```yaml
- name: Configure web server
  hosts: webservers
  tasks:
    - name: Install nginx
      apt:
        name: nginx
        state: present

    - name: Start nginx
      service:
        name: nginx
        state: started
        enabled: yes
```

- Agentless (SSH-based)
- YAML-based declarative syntax

Trade-offs

- ✓ No agent installation required
- ✓ Simple, readable YAML syntax
- ✓ Lower barrier to entry

2010-2014

The Cloud Native Shift

First Generation

- Configure individual machines
- Install and manage software
- Handle system dependencies
- Maintain running services

Key Changes

ZeroMQ → SNS
PostgreSQL → RDS
File Systems → S3
Process Management → Lambda

Second Generation

- Declare cloud resources
- Use managed services
- Higher-level abstractions
- Cloud provider handles operations

Resource-Oriented

Infrastructure defined as a collection of managed resources rather than configuration steps

The focus shifted from "how to run services" to "which services do we need?" ...

2011

CloudFormation

```yaml
Resources:
  WebServerInstance:
    Type: AWS::EC2::Instance
    Properties:
      ImageId: ami-0c55b159cbfafe1f0
      InstanceType: t2.micro
      SecurityGroups:
        - !Ref WebServerSecurityGroup
      UserData:
        Fn::Base64: !Sub |
          #!/bin/bash
          yum update -y
          yum install -y httpd

  WebServerSecurityGroup:
    Type: AWS::EC2::SecurityGroup
    Properties:
      GroupDescription: Enable HTTP access
      SecurityGroupIngress:
        - IpProtocol: tcp
          FromPort: 80
```

- Native AWS Integration
- JSON/YAML Templates
- Built-in State Management

Trade-offs

- ✓ Deep AWS Integration
- ✓ Automatic Rollbacks
- ✗ AWS-only
- ✗ Complex Template Syntax

2014

Terraform

```
provider "aws" {
  region = "us-west-2"
}

resource "aws_instance" "web" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"

  tags = {
    Name = "WebServer"
  }
}

resource "aws_security_group" "allow_http" {
  name        = "allow_http"
  description = "Allow HTTP inbound traffic"

  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
```

- Multi-Cloud Support
- HCL Configuration Language
- Resource Graph & Planning

Trade-offs

- ✅ Provider-agnostic
- ✅ Clear State Management
- ✅ Rich Provider Ecosystem
- ❌ State File Management

2014-2018

Declarative Cloud Challenges

</> Custom DSL Limitations

- No general-purpose programming facilities
- Limited code reuse capabilities
- Difficult to implement complex logic
- Poor developer experience

Verbose Configuration

- Full resource configuration required
- Complex 'glue' configurations (IAM, networking)
- Limited abstraction capabilities
- Repetitive boilerplate code

What Was Needed

- Programming language constructs
- Better abstraction capabilities
- Reusable components
- Modern development workflows

These limitations led to the rise of the next generation of infrastructure as code tooling ...

2019

AWS CDK

```typescript
import * as cdk from 'aws-cdk-lib';
import * as lambda from 'aws-cdk-lib/aws-lambda';
import * as apigateway from 'aws-cdk-lib/aws-apigateway';

export class ApiStack extends cdk.Stack {
  constructor(scope: cdk.App, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Create Lambda function
    const handler = new lambda.Function(this, 'Handler', {
      runtime: lambda.Runtime.NODEJS_18_X,
      code: lambda.Code.fromAsset('lambda'),
      handler: 'handler.main',
    });

    // Create API Gateway
    const api = new apigateway.RestApi(this, 'Api');
    api.root.addMethod('GET',
      new apigateway.LambdaIntegration(handler));
  }
}
```

- TypeScript-first Development
- Constructs Ecosystem
- CloudFormation Under the Hood

Trade-offs

- ✅ Full Programming Language Power
- ✅ Type Safety & IDE Support
- ❌ AWS-only
- ❌ All Cloudformation issues

2018

Pulumi

```
import * as pulumi from "@pulumi/pulumi";
import * as aws from "@pulumi/aws";

const bucket = new aws.s3.Bucket("my-bucket");

const lambdaRole = new aws.iam.Role("lambdaRole", {
    assumeRolePolicy: aws.iam.assumeRolePolicyForPrincipal({
        Service: "lambda.amazonaws.com"
    }),
});

const lambda = new aws.lambda.Function("myLambda", {
    code: new pulumi.asset.AssetArchive({
        ".": new pulumi.asset.FileArchive("./app"),
    }),
    role: lambdaRole.arn,
    handler: "index.handler",
    runtime: "nodejs18.x",
});
```

- Multi-Cloud Native
- </> Multiple Language Support

Trade-offs

- ✅ Provider-agnostic
- ✅ Language Choice Flexibility
- ❌ Learning Curve

2020

SST (Serverless Stack)

```
import { StackContext, Api, Auth } from "@serverless-stack/resources";

export function MyStack({ stack }: StackContext) {
  const auth = new Auth(stack, "auth", {
    login: ["email"],
  });

  const api = new Api(stack, "api", {
    authorizer: "iam",
    routes: {
      "GET  /notes": "functions/list.main",
      "POST /notes": "functions/create.main",
    },
  });

  auth.attachPermissionsToUser(api);

  return {
    api: api.url,
    auth: auth.auth.userPoolId,
```

- Higher-level Abstractions
- Live Lambda Development
- Initially built on AWS CDK, but recently migrated to Pulumi.

Trade-offs

- Higher-Level Abstractions
- Developer Experience Focus

Quick Recap

- **2005-2010: Configuration Management**
  - CFEngine, Puppet, Chef, Ansible
  - Focus on server configuration and management
- **2011-2017: Declarative Cloud**
  - CloudFormation, Terraform, ARM Templates
  - Cloud resource definitions in YAML, JSON, HCL
- **2018-Present: Imperative Cloud**
  - AWS CDK, Pulumi, SST
  - Programming languages for infrastructure

Present Day

The Road Ahead

Current Limitations

- Service-level abstractions only
- Deep cloud service knowledge required
- Infrastructure isolated from application code
- Complex deployment coordination

What We Need

- Business-level abstractions
- Closer integration with application code
- Intent-based infrastructure
- Unified deployment workflows

Moving towards infrastructure that adapts to your application's needs...

Next Generation

</> New Programming Languages

```
bring cloud;

let api = new cloud.Api();
let bucket = new cloud.Bucket();

api.get("/files", inflight (req) => {
  let files = bucket.list();
  return {
    status: 200,
    body: files
  };
});

api.post("/files", inflight (req) => {
  let key = req.body.key;
  let data = req.body.data;
  bucket.put(key, data);
  return {
    status: 201
  };
});
```

Cloud-native by design

Examples

- Wing Language
- Darklang

Trade-offs

- ✅ Purpose-built for cloud
- ✅ Enhanced development experience
- ❌ New language learning curve
- ❌ Limited ecosystem

Next Generation

SDK-Based Approach

```
import { api, storage } from '@nitric/sdk'

// Create a bucket
const files = storage('files').bucket()

// Create an API
api('public').get('/files', async (ctx) => {
  const fileList = await files.list()
  return ctx.ok(fileList)
})

api('public').post('/files', async (ctx) => {
  const { key, data } = ctx.req.body
  await files.file(key).write(data)
  return ctx.created()
})
```

API Abstracted Cloud APIs

Examples

Ampt → Nitric e Encore

Trade-offs

- ✅ Use existing languages
- ✅ Familiar development model
- ❌ Vendor lock-in to SDK
- ❌ Limited community ecosystem

Next Generation

SDK + Annotations

```rust
use axum::{routing::get, Json, Router};
use sqlx::PgPool;

async fn hello_db(
    pool: State<PgPool>
) -> &'static str {
    sqlx::query("SELECT 1")
        .execute(&pool)
        .await
        .unwrap();
}

#[shuttle_runtime::main]
async fn main(
    #[shuttle_shared_db::Postgres] pool: PgPool,
) -> shuttle_axum::ShuttleAxum {
    let router = Router::new()
        .route("/", get(hello_db))
        .with_state(pool);
```

- Rich Development Experience
- Infrastructure Generation

Examples

- Shuttle

Trade-offs

- ✅ Framework-style development
- ❌ Double learning curve
- ❌ More effort to get them to play nice together

Next Generation

Pure Annotations

```
const app = express();

/* @klotho::persist {
 *   id = "petsByOwner"
 * }
 */
const petsByOwner = new Map();

async function addPetName(req, res) {
  const {pet, owner} = req.body;
  await petsByOwner.set(owner, pet);
}

app.post('/pets', addPetName);
/*
 * @klotho::expose {
 *  id = "pet-api"
 *  target = "public"
 *  description = "Exposes the Pet API to the internet"
 * }
```

- Architecture **from** Code
- </> Native SDK Usage

Examples

- Klotho

Trade-offs

- ✓ Minimal learning curve
- ✓ Architecture-level abstractions
- ✗ Limited infrastructure control

Key Trends in Infrastructure as Code

- **Higher Abstractions**
  - From server configs to business logic
  - Increasing levels of abstraction
- **Better Developer Experience**
  - Modern development workflows
  - Rich tooling and IDE support
- **Closer to Application Code**
  - Unified development experience
  - Infrastructure derived from intent

When to Choose What?

Lessons learned the hard way

Small Teams

Is infrastructure a core differentiator?

If no, consider managed platforms, such as:

- Digital Ocean App Platform
- Vercel
- Netlify

Growing Teams

When costs increase:

- Choose tools in your app's language, like CDK or Pulumi

Enterprise

Additional considerations:

- Developer tooling & standards
- Licensing & compliance
- Team training & support
- Multi-cloud strategy

Remember: The best tool is often the one that lets your team ship value fastest ...

Recap

What is IaC?

- Managing infrastructure through code & version control
- Declarative or imperative definitions
- Automated, repeatable deployments

Infrastructure Evolution

- Host Provisioning (2005-2010)
- Declarative Cloud (2011-2017)
- Imperative Cloud (2018+)

Trade-offs & Considerations

- Developer Experience vs Complexity
- Multi-cloud vs Single-provider
- DSL vs Programming Languages

Future Trends

- Business-level abstractions
- Intent-based infrastructure
- Unified deployment workflows

Resources

- **Historical Evolution**
  - "A Brief DevOps History: The Roots of Infrastructure as Code"
  - Comprehensive timeline from Make (1976) to modern tools
- **Technical Deep Dive**
  - "History and Future of Infrastructure as Code"
  - Detailed analysis of IaC generations and their characteristics
- **Next Generation Approaches**
  - "State of Infrastructure from Code 2023"
  - Comprehensive overview of modern IaC approaches
- **Cloud Evolution**
  - "10 Years of Cloud Infrastructure as Code"
  - Focus on serverless and modern cloud patterns

Questions? 😀