Infrastructure as Code Support in Graal CI

JSON-Driven IaC Framework with Pulumi & GitLab CI/CD

Yassine DBAICHI

École Hassania des Travaux Publics

Oracle Labs - GraalVM RISQ Team

Academic Year 2024-2025

Agenda

1. Company & Team Context

Oracle, Oracle Labs, GraalVM
RISQ Team mission and focus

2. Problem & Solution Overview

Current challenges, requirements, and proposed JSON-driven IaC solution

3. Project Management

Agile methodology, communication tools, and development workflow

4. Technical Implementation

Architecture, design patterns, CI/CD integration, and deployment

5. Results & Validations

Implementation results, testing outcomes, and performance metrics

6. Future Enhancements

Current limitations, future work, and conclusion

Company & Team Context

Oracle Corporation

Global Technology Leader: Multinational computer technology corporation specializing in database software and cloud solutions

Founded: 1977 by Larry Ellison, Bob Miner, and Ed Oates

Headquarters: Austin, Texas, USA (Oracle Headquarters)

Market Position: One of the largest enterprise software companies worldwide

Core Products: Oracle Database, Oracle Cloud Infrastructure (OCI), Enterprise Applications

Revenue: $50+ billion annually with 430,000+ employees globally

Oracle Labs - Research & Innovation

Mission: Conduct research to advance the state of the art in computer science and systems

Focus Areas:

- Programming languages & compilers
- Virtual machines & runtime systems
- Database systems & optimization
- Cloud infrastructure & security

Key Projects: GraalVM, Truffle Framework, SubstrateVM

Research Excellence

Oracle Labs bridges cutting-edge academic research with real-world enterprise applications

Output: Open-source projects, academic publications, and production-ready technologies

Collaboration

Strong partnerships with universities and research institutions worldwide

GraalVM - High-Performance Polyglot Runtime

What is GraalVM? A universal virtual machine for running applications written in JavaScript, Python, Ruby, R, JVM languages, and more

Key Features:

- Polyglot capabilities (multiple languages)
- Native Image compilation
- Superior performance optimization
- Reduced memory footprint

Use Cases: Microservices

GraalVM RISQ Team

Team Name: RISQ (Release Infrastructure, Systems & Quality)

Mission: Ensure reliability, scalability, and quality of GraalVM infrastructure and development workflows

Responsibilities:

- CI/CD pipeline management and optimization
- Infrastructure automation and provisioning
- Testing infrastructure maintenance
- Developer tooling and experience improvements

Tech Stack: GitLab CI/CD, Oracle Cloud Infrastructure, Python, Bash scripting

The RISQ team acts as the backbone supporting GraalVM development, ensuring smooth operations for hundreds of

Project Context & Internship Focus

Team Challenge

GraalVM requires frequent infrastructure provisioning for testing, CI/CD, and development environments across multiple cloud regions

Manual Bottleneck

Infrastructure management was manual, error-prone, and time-consuming - limiting team agility

Project Goal

Design and implement an automated Infrastructure as Code (IaC) solution integrated with Graal CI pipeline

My Role

Lead developer responsible for architecture design, implementation, and GitLab CI/CD integration

Problem & Solution Overview

Current Infrastructure Management Challenges

Manual Console Operations: Team members had to manually create resources through OCI web console - time-consuming and repetitive

Lack of Version Control: No tracking of infrastructure changes, making rollbacks and audits difficult

Configuration Drift: Resources created manually often diverged from documented standards

Scalability Issues: Provisioning multiple similar environments was labor-intensive

Knowledge Silos: Infrastructure knowledge concentrated in few team members

No CI/CD Integration: Infrastructure provisioning was separate from development workflow

Result: Slow deployment cycles, increased error rates, and reduced

Functional Requirements

FR1 - Declarative Configuration: Users define infrastructure in human-readable format (JSON) without writing Python/Terraform code

FR2 - Resource Support: Support for essential OCI resources (VCN, Subnets, Compute Instances, Storage Buckets, Security Lists, Routing)

FR3 - Dependency Management: Automatic resolution of resource dependencies (e.g., subnet requires VCN, instance requires subnet)

FR4 - CI/CD Integration: Seamless integration with GitLab CI/CD pipeline (plan, preview, apply stages)

FR5 - State Management: Track infrastructure state to enable updates and deletions

FR6 - Multi-Environment: Support for development, staging, and production configurations

Non-Functional Requirements

Performance

- Provision infrastructure in <5 minutes
- Support concurrent deployments
- Minimal API calls to OCI

Security

- Credentials stored securely (GitLab secrets)
- No hardcoded sensitive data
- Audit trail via Git commits

Extensibility

- Plugin architecture for new resources
- Support for future cloud providers
- Custom validation rules

IaC Technology Selection: Terraform vs Pulumi

Terraform

Pros:

- Industry standard with mature ecosystem
- HCL declarative language
- Large provider ecosystem
- Strong community support

Cons:

- Custom DSL (HCL) requires learning
- Limited programming logic & abstraction
- State management complexity
- Hard to build dynamic configurations

Pulumi ✓ (Selected)

Pros:

- Use real programming languages (Python)
- Full programmatic control & abstraction
- Built-in state management
- Excellent OCI support
- Enables framework development on top
- Team Python expertise leverage

Why chosen: Perfect fit for building JSON-driven abstraction layer. Pulumi's programmatic approach allows us to create a framework that hides IaC complexity from users.

How does Pulumi work?

1. Write Code

Define infrastructure using Python - loops, functions, conditionals available

2. Preview

Run 'pulumi preview' to see what will change before applying

3. Deploy

Pulumi SDK translates Python to OCI API calls and provisions resources

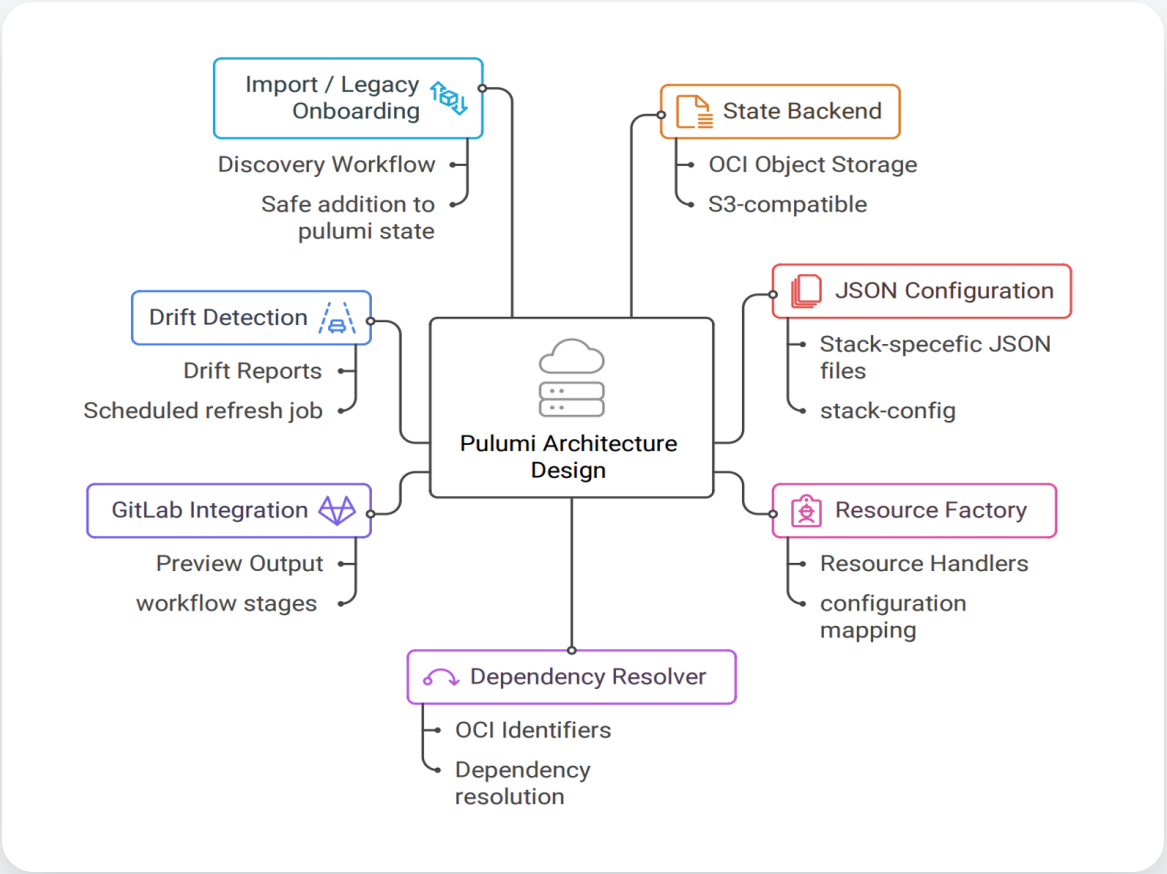
4. State Management

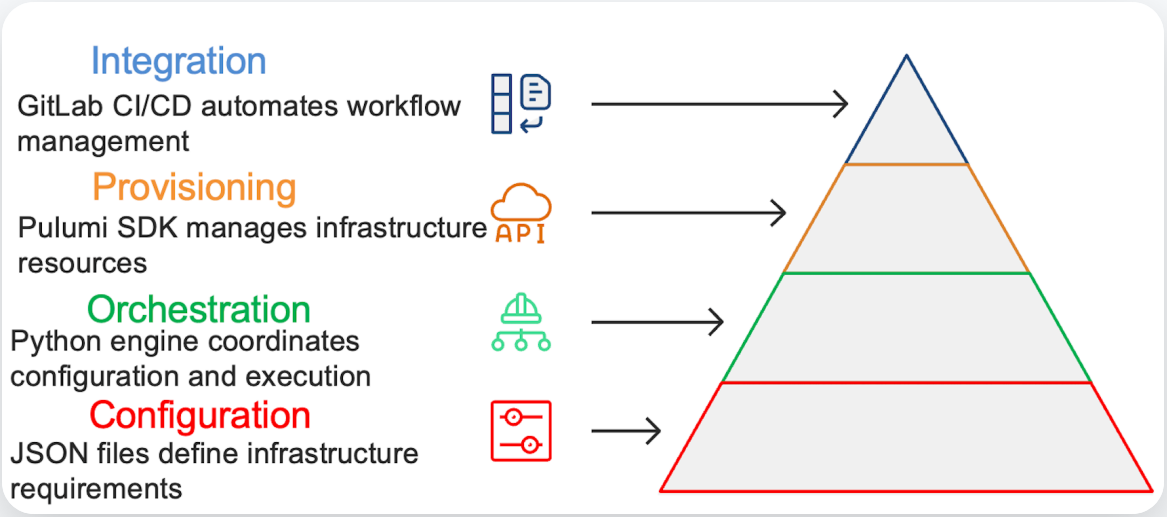
Pulumi tracks all resources in state file (OCI Object Storage)

Python-First: Leverage existing team Python expertise

OCI Provider: First-class Oracle Cloud Infrastructure support

Framework Foundation: Enables building JSON-driven abstraction layer on top


Import / Legacy Onboarding
Discovery Workflow
Safe addition to pulumi state
State Backend
OCI Object Storage
S3-compatible
Drift Detection
Drift Reports
Scheduled refresh job
JSON Configuration
Stack-specefic JSON files
stack-config
Pulumi Architecture Design
GitLab Integration
Preview Output
workflow stages
Resource Factory
Resource Handlers
configuration mapping
Dependency Resolver
OCI Identifiers
Dependency resolution


Integration
GitLab CI/CD automates workflow management
Provisioning
Pulumi SDK manages infrastructure resources
API
Orchestration
Python engine coordinates configuration and execution
Configuration
JSON files define infrastructure requirements

Project Management

Project Management Approach

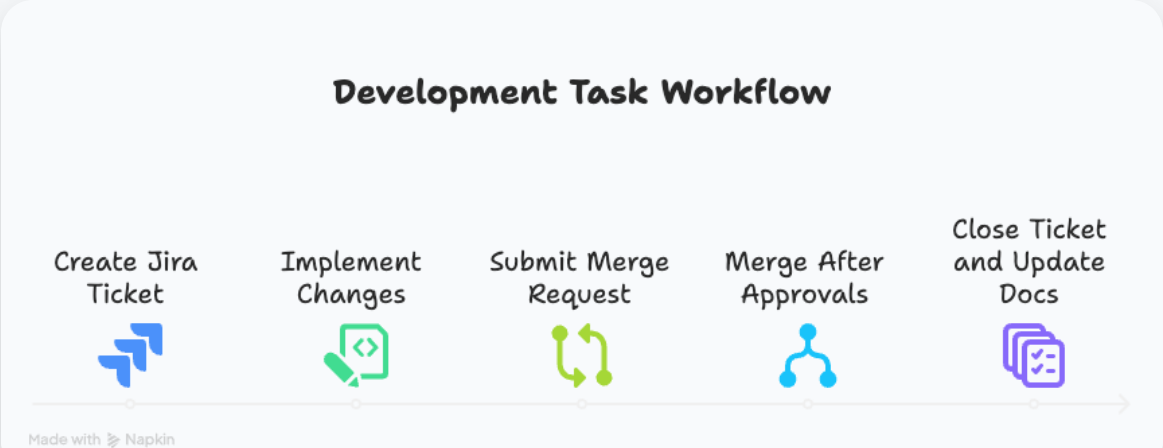
Methodology: Kanban Agile - visual workflow management with continuous delivery

Project Tracker: Task board for tracking work items (To Do, In Progress, Done)

Regular Meetings: Weekly sync meetings with mentor to review progress and blockers

Continuous Flow: Work items pulled as capacity allows, no fixed sprint boundaries

Focus on Delivery: Prioritize completing tasks over starting new ones


Development Task Workflow
Create Jira Ticket
Implement Changes
Submit Merge Request
Merge After Approvals
Close Ticket and Update Docs
Made with Napkin

Communication & Collaboration Tools

Slack

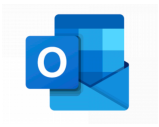
Daily communication, quick questions, team updates

Zoom

Weekly sync meetings, architecture discussions

Jira

Task tracking, sprint planning, progress monitoring



Outlook

Meeting scheduling, formal communications

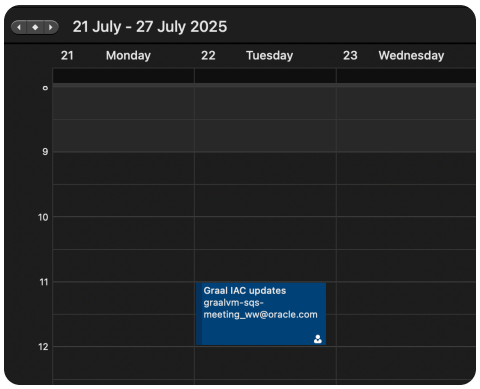
Confluence

Documentation, design specs, knowledge base

GitLab

Code repository, CI/CD, merge requests

Mentor Check-ins & Team Meetings



Weekly Sync Meetings

Frequency: Every Tuesday 11:00 AM
Duration: 45-60 minutes
Agenda: Progress review, blockers discussion, next steps planning
Participants: Intern + mentor (Hamza Ghaissi)

Ad-hoc Sessions

Technical architecture discussions
Code reviews and pair programming
Debugging complex issues
Design decision consultations

Communication Style: Open-door policy via Slack - quick responses and collaborative problem-solving

Project Timeline & Key Milestones

Phase 1: Research (2 weeks)

- OCI API exploration
- Pulumi SDK study
- Requirements gathering
- Architecture design

Phase 2: Prototype (2 weeks)

- Core framework skeleton
- VCN + Instance resources
- Factory pattern POC
- Basic dependency resolver

Phase 3: Development (4 weeks)

- All 7 resource types
- JSON configuration loader
- Dependency resolution logic
- Error handling

Phase 4: CI/CD Integration (2 weeks)

- GitLab pipeline config
- Plan & deploy stages
- Manual

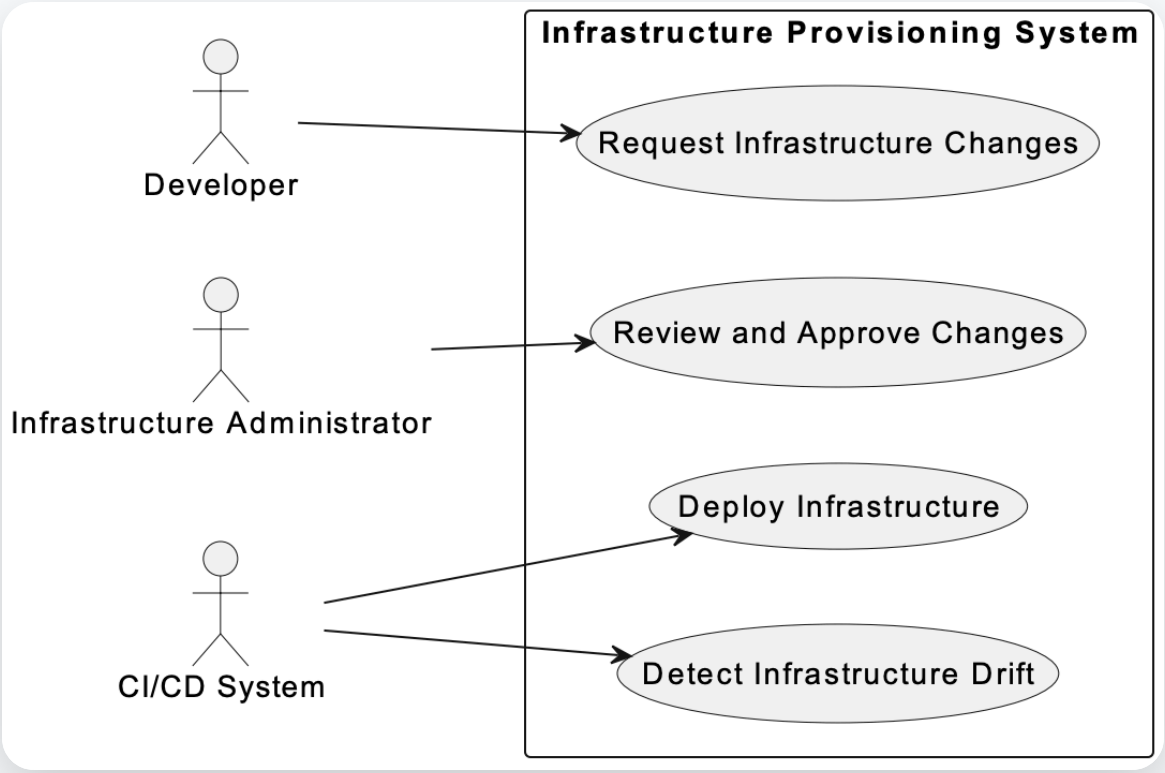
Phase 5: Testing (2 weeks)

- Integration testing
- Edge case validation
- Performance testing
- Bug fixes

Phase 6: Documentation (1 week)

- User guide creation
- Code documentation
- Architecture diagrams
- Final

Technical Implementation


Infrastructure Provisioning System
Developer
Request Infrastructure Changes
Infrastructure Administrator
Review and Approve Changes
Deploy Infrastructure
CI/CD System
Detect Infrastructure Drift

```
Initial-Pulumi-Setup/
├── Pulumi.yaml                # Pulumi project and stack metadata
├── Pulumi.dev.yaml            # Dev stack settings (config...)
├── configs/                   # Stack-specific configuration files
│   ├── dev.json              # Development stack resources
│   ├── prod.json             # Production stack resources
│   └── test.json             # Test stack resources
├── requirements.txt           # Python dependencies
├── venv/                      # Python virtual environment
├── .env.example               # Template for environment variables
├── .env.local                 # Your personal environment variables (create this)
├── core/                      # Base_resource, factory, resolver
├── loaders/                   # JSON loader and .env reader
├── resources/                 # Python classes for each OCI resource type
└── main.py                    # Entrypoint: one-loop provisioning logic
```

JSON Configuration Example

infrastructure.json

```
{
  "name": "production-vcn",
  "cidrBlock": "10.0.0.0/16",
  "internetGateways": [...]
  "securityLists": [...]
  "routeTables": [...]
  "subnets": [...]
}
```

Name-Based References: Resources reference each other by human-readable names

Automatic Resolution: Framework resolves names to OCIDs automatically

Core Framework Components

BaseResource (Abstract)

File: core/base_resource.py (21 lines)
Purpose: Abstract base class defining resource interface
Methods:

- create() - Factory method
- build() - Abstract (must implement)

ResourceFactory

File: core/resource_factory.py (38 lines)
Purpose: Factory pattern for dynamic instantiation
Registry: Maps type strings → Python classes

7 resource types registered

DependencyResolver

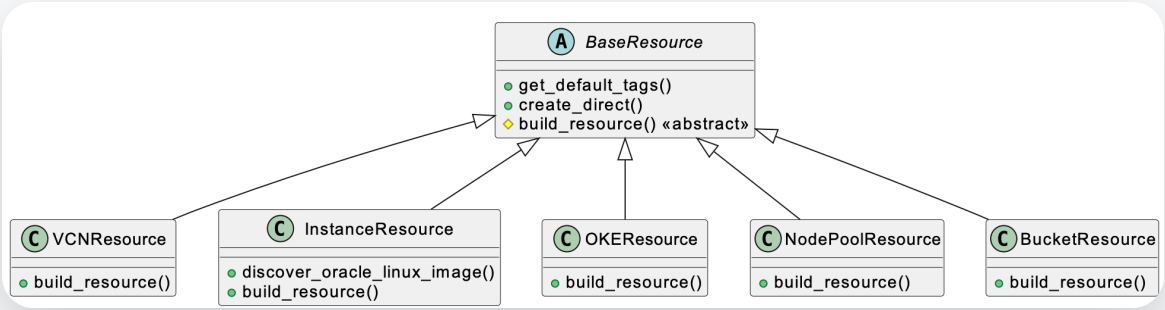
File: core/dependency_resolver.py (14 lines)
Purpose: Resource reference manager
Methods:

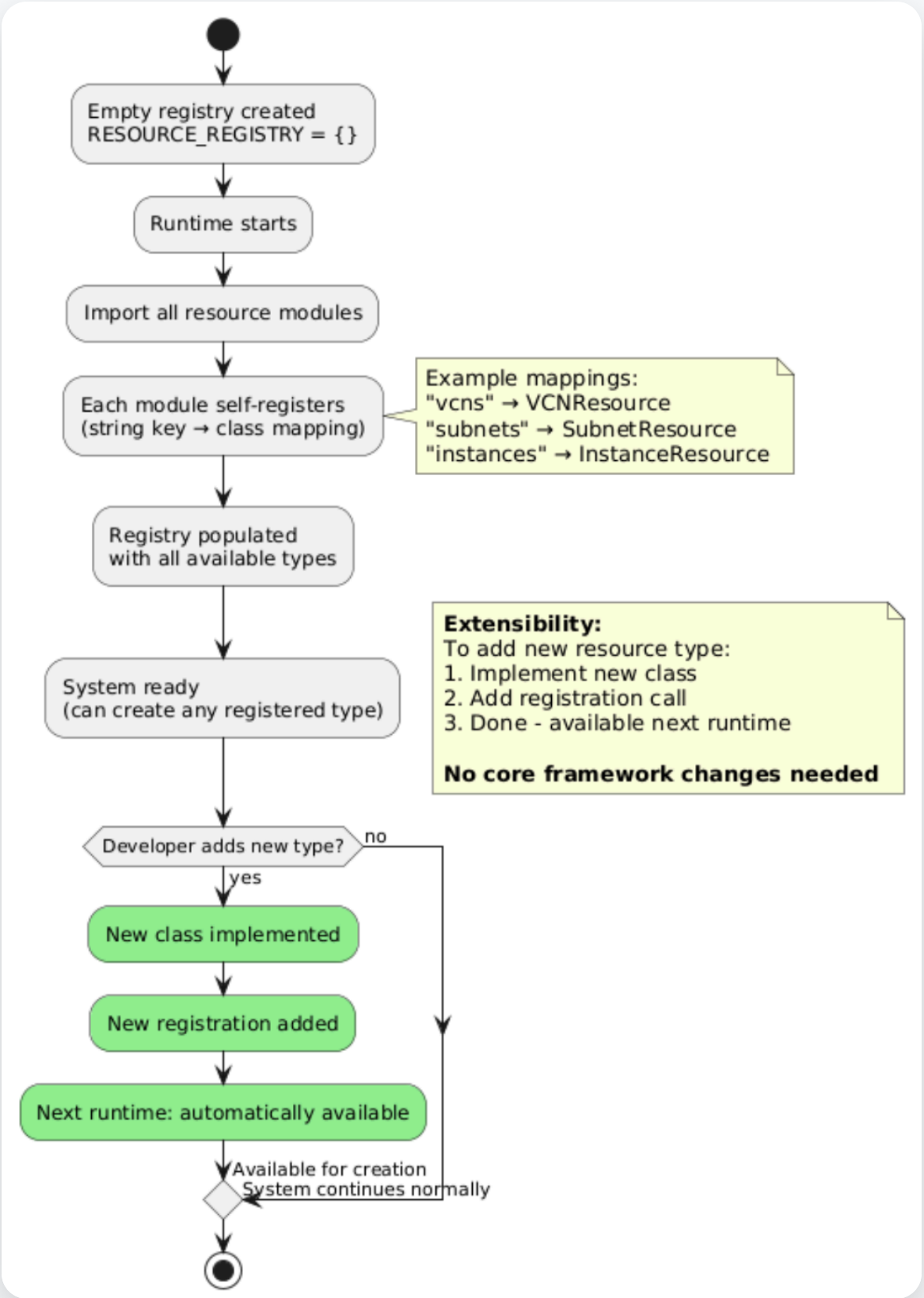
- register_resource(name, resource)
- get_resource_id(name)

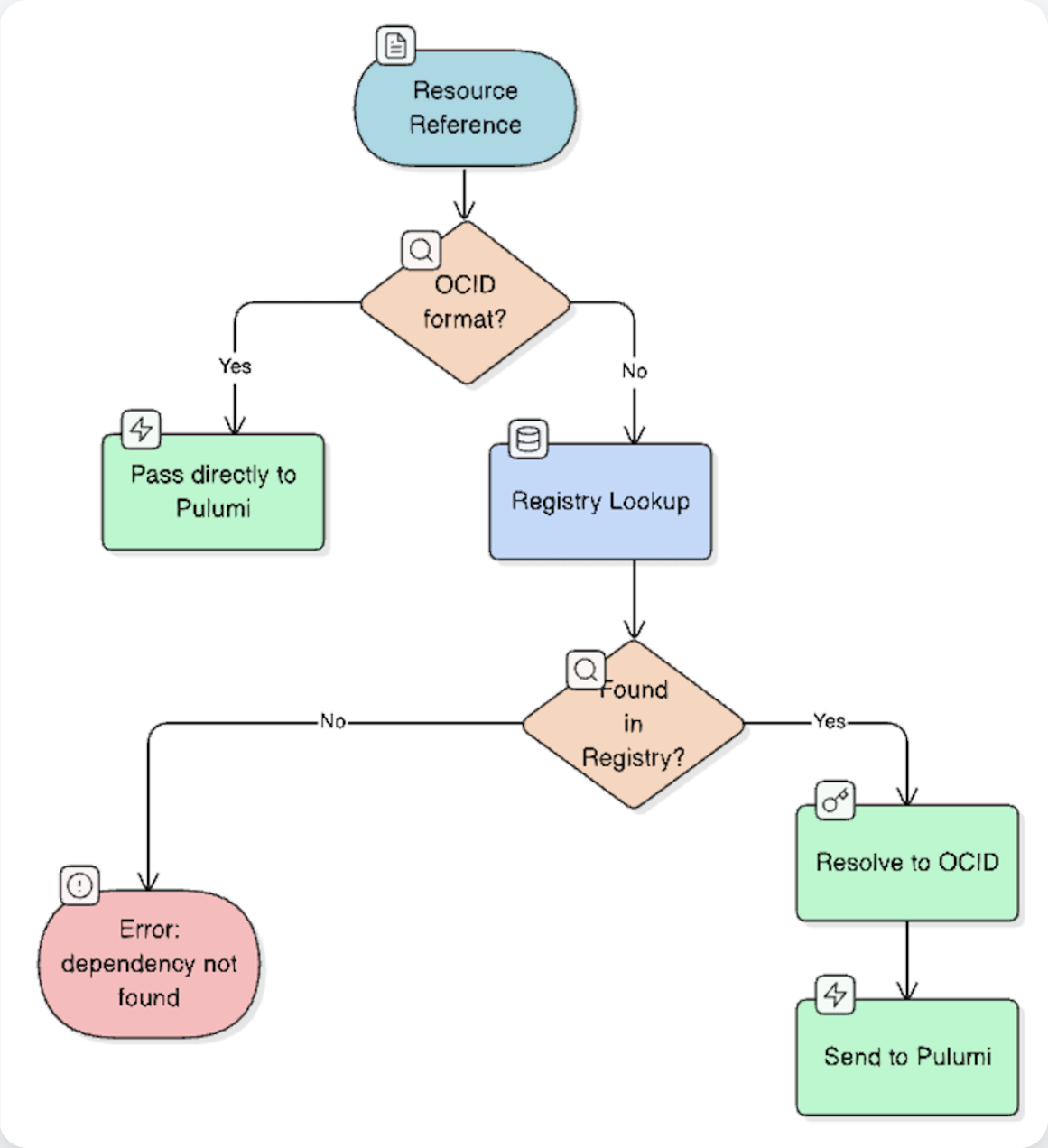
ConfigLoader

File: config/config_loader.py (25 lines)
Purpose: Parse infrastructure.json
Methods:

- get_vcns_config()
- get_instances_config()
- get_image_config()


BaseResource
get_default_tags()
create_direct()
build_resource() «abstract»
VCNResource
build_resource()
InstanceResource
discover_oracle_linux_image()
build_resource()
OKEResource
build_resource()
NodePoolResource
build_resource()
BucketResource
build_resource()


Empty registry created
RESOURCE_REGISTRY = {}
Runtime starts
Import all resource modules
Each module self-registers
(string key → class mapping)
Example mappings:
"vcns" → VCNResource
"subnets" → SubnetResource
"instances" → InstanceResource
Registry populated
with all available types
System ready
(can create any registered type)
Extensibility:
To add new resource type:
1. Implement new class
2. Add registration call
3. Done - available next runtime
No core framework changes needed
Developer adds new type?
no
yes
New class implemented
New registration added
Next runtime: automatically available
Available for creation
System continues normally


Resource Reference
OCID format?
Yes
No
Pass directly to Pulumi
Registry Lookup
Found in Registry?
No
Yes
Error: dependency not found
Resolve to OCID
Send to Pulumi

Implemented Resource Types

1. VCN (Virtual Cloud Network)

Purpose: Foundation network container

2. Internet Gateway

Purpose: Enable internet connectivity

3. Security List

Purpose: Firewall rules (ingress/egress)

4. Route Table

Purpose: Network routing configuration

5. Subnet

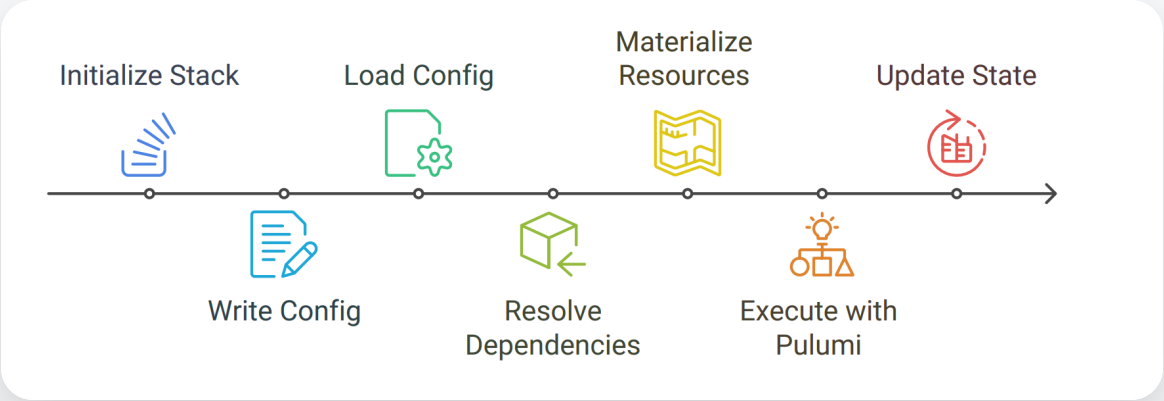
Purpose: Network segment within VCN

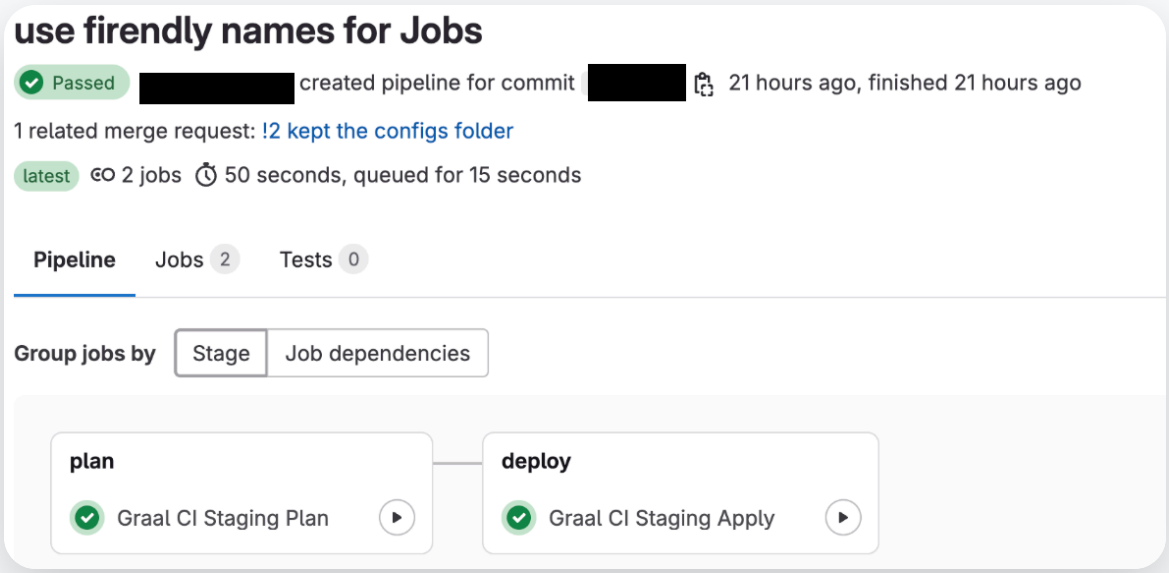
6. Compute Instance

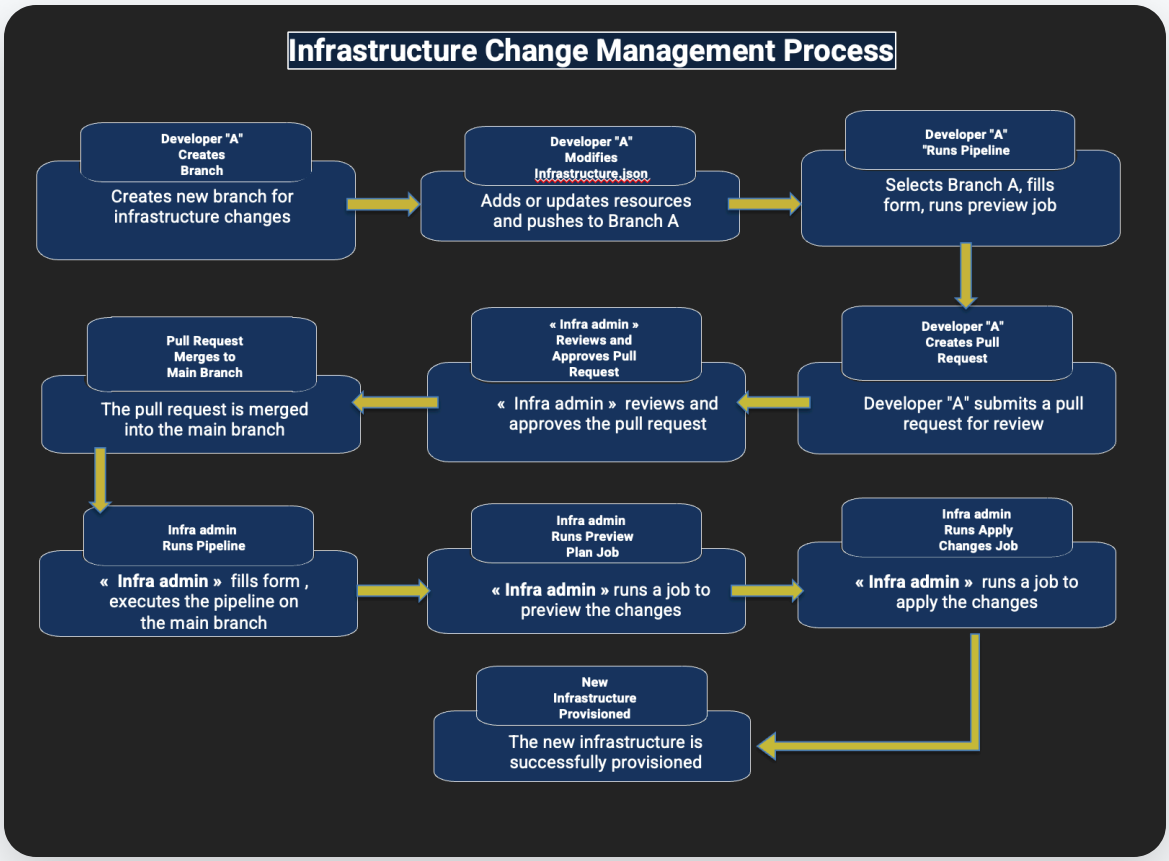
Purpose: Virtual machine

7. Object Storage Bucket

Purpose: S3-compatible object storage


Initialize Stack
Load Config
Materialize Resources
Update State
Write Config
Resolve Dependencies
Execute with Pulumi


use firendly names for Jobs
Passed
[redacted] created pipeline for commit [redacted] 21 hours ago, finished 21 hours ago
1 related merge request: !2 kept the configs folder
latest
2 jobs
50 seconds, queued for 15 seconds
Pipeline
Jobs 2
Tests 0
Group jobs by
Stage
Job dependencies
plan
Graal CI Staging Plan
deploy
Graal CI Staging Apply


Infrastructure Change Management Process
Developer "A" Creates Branch
Creates new branch for infrastructure changes
Developer "A" Modifies Infrastructure.json
Adds or updates resources and pushes to Branch A
Developer "A" "Runs Pipeline
Selects Branch A, fills form, runs preview job
Developer "A" Creates Pull Request
Developer "A" submits a pull request for review
« Infra admin » Reviews and Approves Pull Request
« Infra admin » reviews and approves the pull request
Pull Request Merges to Main Branch
The pull request is merged into the main branch
Infra admin Runs Pipeline
« Infra admin » fills form , executes the pipeline on the main branch
Infra admin Runs Preview Plan Job
« Infra admin » runs a job to preview the changes
Infra admin Runs Apply Changes Job
« Infra admin » runs a job to apply the changes
New Infrastructure Provisioned
The new infrastructure is successfully provisioned

Results & Validations

Implementation Results

✅ Deliverables Completed

- Python framework (800 LOC)
- 7 OCI resource types
- JSON configuration system
- GitLab CI/CD pipeline
- Documentation & diagrams

🎯 Requirements Met

- ✓ Declarative JSON config
- ✓ Dependency resolution
- ✓ CI/CD integration
- ✓ Multi-environment support
- ✓ State management

🚀 Production Ready

- Error handling implemented
- Tested on dev & staging
- Documentation complete
- Team training delivered
- Now in active use by RISQ team

```
└─ pulumi up
Previewing update (graalci_prod):
     Type                          Name                      Plan
 +   pulumi:pulumi:Stack           graal-iac-graalci_prod    create
 +   ├─ oci:Core:Vcn               production-vcn            create
 +   ├─ oci:Core:RouteTable        prod-rt                   create
 +   ├─ oci:Core:Subnet            prod-subnet               create
 +   ├─ oci:Core:InternetGateway   prod-igw                  create
 +   ├─ oci:Core:SecurityList      prod-security             create
 +   ├─ oci:ObjectStorage:Bucket   prod-data-bucket          create
 +   └─ oci:Core:Instance          prod-app                  create

Outputs:
    prod-app_public_ip: [unknown]

Resources:
    + 8 to create
```

```
Do you want to perform this update? yes
Updating (graalci_prod):
     Type                          Name                      Status
 +   pulumi:pulumi:Stack           graal-iac-graalci_prod    created (46s)
 +   ├─ oci:ObjectStorage:Bucket   prod-data-bucket          created (0.60s)
 +   ├─ oci:Core:Vcn               production-vcn            created (1s)
 +   ├─ oci:Core:InternetGateway   prod-igw                  created (1s)
 +   ├─ oci:Core:SecurityList      prod-security             created (1s)
 +   ├─ oci:Core:RouteTable        prod-rt                   created (1s)
 +   ├─ oci:Core:Subnet            prod-subnet               created (2s)
 +   └─ oci:Core:Instance          prod-app                  created (36s)

Outputs:
    prod-app_public_ip: "152.70.52.111"

Resources:
    + 8 created

Duration: 48s
```

```
└─ ssh -i ~/.ssh/oci_key opc@152.70.52.111
The authenticity of host '152.70.52.111 (152.70.52.111)' can't be established.
ED25519 key fingerprint is SHA256:Yz4gYLSbkaL5AKMLxz0T6PW2yCer0vo4BXbmcHdE6zE.
This key is not known by any other names.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '152.70.52.111' (ED25519) to the list of known hosts.
Activate the web console with: systemctl enable --now cockpit.socket

[opc@ProductionAppServer ~]$ uname -a
Linux ProductionAppServer 5.15.0-312.187.5.1.el8uek.x86_64 #2 SMP Mon Sep 8 12:49:36 PDT 2025 x86_64 x86_64 x86_64 GNU/Linux
[opc@ProductionAppServer ~]$ exit
logout
Connection to 152.70.52.111 closed.
```

Testing & Validation Results

Unit Testing

Scope: Individual resource classes

- Factory pattern registration
- Dependency resolver lookup
- Config loader parsing

✓ All components working correctly

Integration Testing

Test Cases:

- Single VCN + 1 instance
- 2 VCNs + 2 instances
- Complex network (IGW + RT + SL)
- Bucket creation

✓ End-to-end provisioning successful

CI/CD Pipeline Testing

Validated:

- Plan stage (preview changes)
- Deploy stage (apply changes)
- Manual approval gate
- State persistence

✓ Pipeline stable over 20+ runs

Edge Cases Tested

- ✓ Missing dependency (error caught)
- ✓ Invalid CIDR block (validation)
- ✓ Duplicate resource names (detected)
- ✓ Image not compatible with shape

Performance Metrics & Improvements

95%

Time Reduction

From 2-3 hours manual setup
To <5 minutes automated

100%

Consistency

Zero configuration drift
All envs from same templates

85%

Error Reduction

Automated validation
Dependency checking

Before (Manual)

Average setup: 2.5 hours
Error rate: ~15%
Documentation: Often outdated
Reproducibility: Low

After (Automated)

Average setup: 4 minutes
Error rate: <2%
Documentation: Self-documenting JSON
Reproducibility: 100%

Business Impact & Team Benefits

Developer Productivity: Team members can now focus on GraalVM development instead of infrastructure management - estimated 10+ hours saved per week across the team

Faster Testing Cycles: Spin up test environments in minutes instead of hours - enables rapid experimentation and bug fixes

Onboarding Acceleration: New team members can provision infrastructure on day one without specialized training - reduces onboarding time from days to hours

Cost Optimization: All infrastructure tracked in Git - easier to identify and delete unused resources, preventing budget overruns

Future Enhancements

Future Improvements

More Resource Coverage

Expand framework to support additional OCI resources like Load Balancers, Databases, and File Storage

Schema Validation

Implement JSON schema validation to catch configuration errors before deployment

Role Identification

Define clear roles within the team on how to use the framework and establish best practices

Future Work & Enhancement Roadmap

Q1 2026: Extended Resources

Priority: High

- Add Load Balancer support
- Autonomous Database provisioning
- File Storage Service integration
- Network Load Balancer

Effort: 4 weeks

Q2 2026: Validation & Safety

Priority: High

- JSON schema validation
- Pre-deployment cost estimation
- Security policy enforcement
- CIDR conflict detection

Effort: 3 weeks

Q3 2026: Operational Features

Priority: Medium

- Automated drift detection
- Scheduled state refresh jobs
- Notification integration (Slack)
- Resource tagging automation

Effort: 5 weeks

Q4 2026: Multi-Cloud Expansion

Priority: Low

- AWS provider support
- Azure provider support
- Provider abstraction layer
- Cross-cloud networking

Effort: 8 weeks

Lessons Learned & Best Practices

Technical Insights

Start Simple: MVP with 2 resources, then expand - avoid over-engineering upfront

Embrace Patterns: Factory + Resolver patterns made code extensible and maintainable

Test Early: Integration testing caught dependency issues before production

Document as You Go: Inline comments and diagrams saved time later

Process Insights

Agile Works: 2-week sprints with demos kept momentum high

Feedback Loop: Weekly mentor sessions caught issues early

User-Centric: Involving team in design ensured adoption

Challenges Overcome

Dependency Hell: Solved with centralized resolver pattern

State Management: Pulumi's built-in state backend simplified this

Error Handling: Graceful failures with clear messages improved UX

CI/CD Integration: Manual approval gate provided safety net

Advice for Similar Projects

✓ Prototype quickly, iterate based on feedback

✓ Invest in good abstractions early

✓ Automate testing from day one

✓ Keep code modular (<100 LOC per file)

Conclusion

Successfully delivered a production-ready IaC framework that transforms infrastructure management for the GraalVM RISQ team

🎯 Objectives Achieved

- ✓ JSON-driven configuration
- ✓ 7 OCI resource types
- ✓ GitLab CI/CD integration
- ✓ 95% time reduction
- ✓ Production deployment

💡 Innovation Highlights

Name-based dependency resolution
Factory pattern for extensibility
Declarative JSON interface
Two-stage approval workflow
Self-documenting architecture

📈 Measurable Impact

10+ hours saved per week
Zero configuration drift
85% error reduction
100% reproducibility
Active team adoption

🔮 Future Potential

Expand to 15+ resource types
Multi-cloud support
Advanced validation
Cost optimization features
Enterprise-wide adoption

Acknowledgments

Special Thanks

Hamza GHAISSI - Oracle Labs Mentor & Technical Supervisor

For invaluable guidance, code reviews, and architectural insights throughout the project

Academic Supervisor

Pr. Hatim LECHGAR
École Hassania des Travaux Publics
For academic guidance and project oversight

GraalVM RISQ Team

For welcoming me to the team, providing feedback, and actively using the framework

Oracle Labs & EHTP

For providing this incredible learning opportunity and fostering innovation in infrastructure automation

References & Resources

Technical Documentation

Pulumi Documentation
pulumi.com/docs
Oracle Cloud Infrastructure API Reference
docs.oracle.com/iaas/api
GitLab CI/CD Documentation
docs.gitlab.com/ee/ci
Python Design Patterns
refactoring.guru/design-patterns

Project Resources

GitHub Repository:
github.com/y-dbaichi/pfe-pulumi
Internal Confluence Documentation
Architecture Diagrams (25 PNG files)

Key Technologies

Pulumi: v3.0.0+ (IaC framework)
Python: 3.13 (programming language)
OCI Provider: v2.0.0+ (cloud provider)
GitLab CI/CD: (automation platform)
JSON: (configuration format)

Contact Information

Yassine DBAICHI
Infrastructure Automation Engineer Intern
Oracle Labs - GraalVM RISQ Team
École Hassania des Travaux Publics

Thank You

Questions & Discussion

Yassine DBAICHI

Infrastructure as Code Support in Graal CI

Oracle Labs - GraalVM RISQ Team

Academic Year 2024-2025