# Deployment Strategy - OCaml to Python Migration

## Project Overview

This document outlines the deployment strategy for migrating GeneWeb from OCaml to Python while maintaining system reliability and ensuring zero-downtime deployments.

### Migration Goals

- **Binary Migration**: Replace OCaml binaries with Python equivalents
- **Web Server Migration**: Transition OCaml web server to Python
- **Quality Assurance**: Maintain functionality through comprehensive testing
- **Infrastructure**: Deploy using simple VMs on Google Cloud Platform with Terraform

## Architecture Overview

The deployment strategy follows a three-tier approach: Development branches feed into the Master branch, which then deploys to Production. Each stage includes comprehensive testing with automatic fallback mechanisms to ensure system stability.

## Deployment Pipeline

### Phase 1: Development & Testing

**1.1 Feature Development** The development phase focuses on creating Python equivalents for existing OCaml binaries while maintaining full backward compatibility. Each Python component is developed alongside comprehensive tests to ensure functional parity with the original OCaml implementation.

**1.2 Quality Gates** The testing strategy implements a multi-layered approach starting with unit tests for individual Python components, followed by integration tests for system interactions. Golden Master tests compare outputs between OCaml and Python versions to ensure identical behavior. Performance and security tests validate that Python implementations meet or exceed OCaml benchmarks.

### Phase 2: Continuous Integration

**2.1 GitHub Actions Workflow** The CI/CD pipeline automatically triggers on pushes to master or main branches, running the complete test suite including unit tests, quality checks, Golden Master comparisons, performance benchmarks,

and security scans. This ensures every change is thoroughly validated before deployment.

**2.2 Test Strategy** Golden Master tests serve as the primary validation mechanism, comparing OCaml and Python outputs to ensure identical functionality. Unit tests validate individual components, while integration tests verify system-wide interactions. Performance tests ensure Python implementations maintain acceptable speed, and quality tests enforce code coverage and security standards.

**Phase 3: Master Branch Strategy**

**3.1 Automated Decision Making** The system automatically decides whether to use Python or OCaml binaries based on comprehensive test results. If all tests pass and performance meets benchmarks, Python binaries are selected. Any test failures or performance regressions trigger an automatic fallback to the proven OCaml implementation.

**3.2 Fallback Mechanism** The deployment strategy prioritizes Python implementations as the primary choice while maintaining OCaml binaries as a reliable fallback. Decision criteria include Golden Master test results, performance benchmarks, integration test outcomes, and real-time error rates to ensure system stability.

## Docker Strategy

### 4.1 Multi-Stage Docker Build

The Docker build process uses multi-stage builds to create containers that include both Python and OCaml implementations. This approach allows for runtime decision-making about which implementation to use based on test results and system health.

### 4.2 Container Strategy

The containerization strategy employs base images optimized for both Python and OCaml environments. Health checks are integrated to monitor application status continuously. The system supports graceful degradation by automatically switching to OCaml implementations when Python versions encounter issues. Resource management is optimized to ensure efficient memory and CPU utilization across both implementations.

## Google Cloud Platform Deployment

### 5.1 Simple Infrastructure Components

The GCP infrastructure uses a simplified approach with a VPC network containing public subnets for web servers and private subnets for database access.

Compute Engine VMs serve as the primary deployment targets, with separate staging and production instances. An HTTP(S) Load Balancer provides high availability and traffic distribution. The Artifact Registry stores Docker images and application packages, while Cloud SQL provides managed database services. Comprehensive monitoring and logging are implemented through Cloud Monitoring, Cloud Logging, and Uptime Checks.

### 5.2 Infrastructure as Code with Terraform

Terraform manages the entire infrastructure deployment through modular configurations. The VPC module creates the network foundation, while the compute module provisions virtual machines with appropriate specifications. The load balancer module ensures high availability, and the database module sets up managed Cloud SQL instances. This approach ensures consistent, reproducible deployments across environments.

### 5.3 VM Configuration

The infrastructure uses two primary VM configurations: a staging environment with moderate resources for testing (2 vCPU, 4GB RAM) and a production environment with enhanced specifications (4 vCPU, 16GB RAM). Both run Ubuntu 22.04 LTS with Docker, systemd services, and monitoring agents. The database uses Cloud SQL PostgreSQL with appropriate sizing for the application's needs.

### 5.4 VM Setup and Services

Virtual machines are automatically configured with Docker for containerization, monitoring agents for observability, and systemd services for application management. The Python service runs on the primary port while the OCaml fallback service operates on an alternative port, allowing for seamless switching between implementations based on system health and test results.

## Deployment Workflow

### 6.1 Automated Deployment Pipeline

The deployment process begins when code is pushed to the master branch, automatically triggering the CI/CD pipeline. The system runs a complete test suite including Golden Master tests, builds Docker images for both Python and OCaml implementations, and pushes them to the Artifact Registry. New versions are first deployed to the staging VM for validation through smoke tests, then promoted to production with load balancer health check updates and continuous monitoring.

### 6.2 Simple VM Deployment Strategy

- **Staging VM**: Test new deployments before production

- **Production VM**: Single instance with auto-restart capabilities
- **Load Balancer**: HTTP(S) Load Balancer for high availability
- **Health Monitoring**: Uptime checks and automatic failover
- **Deployment Method**: Docker containers on VMs with systemd services

## Monitoring & Observability

### 7.1 Key Metrics

The monitoring strategy tracks comprehensive performance metrics including response times at various percentiles, system throughput, error rates, and resource utilization. Database performance and user experience metrics provide insights into system health. Business metrics focus on feature adoption rates between Python and OCaml implementations, system reliability measurements, migration progress tracking, and cost optimization achievements.

### 7.2 Alerting Strategy

- **Critical Alerts**: System failures, high error rates
- **Warning Alerts**: Performance degradation, resource limits
- **Info Alerts**: Deployment status, migration milestones

## Security & Compliance

### 8.1 Security Measures

- **Container Scanning**: Vulnerability assessment for all images
- **Network Security**: VPC isolation, firewall rules
- **Access Control**: IAM policies, service accounts
- **Secrets Management**: Google Secret Manager integration
- **Audit Logging**: Comprehensive activity tracking

### 8.2 Compliance

- **Data Protection**: GDPR compliance for genealogical data
- **Backup Strategy**: Automated backups with point-in-time recovery
- **Disaster Recovery**: Multi-region deployment capability

## Configuration Management

### 9.1 Environment Configuration

Environment configurations are tailored for each deployment stage. Development environments enable Python implementations with OCaml fallback and debug mode for troubleshooting. Staging environments maintain Python enablement with fallback capabilities and enhanced performance monitoring. Production environments use automatic Python enablement based on test results while

ensuring OCaml fallback is always available, with comprehensive monitoring across all systems.

### 9.2 Feature Flags

- **Gradual Rollout**: Progressive Python adoption
- **A/B Testing**: Compare Python vs OCaml performance
- **Emergency Switches**: Instant fallback mechanisms

## Migration Phases

**Phase 1: Foundation (Weeks 1-4)**

- ☐ Set up CI/CD pipeline
- ☐ Implement Golden Master tests
- ☐ Create Docker infrastructure
- ☐ Set up GCP environment with Terraform

**Phase 2: Core Migration (Weeks 5-12)**

- ☐ Migrate critical binaries
- ☐ Implement web server in Python
- ☐ Comprehensive testing suite
- ☐ Performance optimization

**Phase 3: Production Deployment (Weeks 13-16)**

- ☐ Staging environment validation
- ☐ Blue-green deployment setup
- ☐ Production rollout with monitoring
- ☐ Performance tuning and optimization

**Phase 4: Optimization (Weeks 17-20)**

- ☐ Remove OCaml dependencies (where stable)
- ☐ Performance improvements
- ☐ Cost optimization
- ☐ Documentation and knowledge transfer

## Troubleshooting & Rollback

### 10.1 Rollback Scenarios

1. **Automatic Rollback**: Failed health checks, high error rates
2. **Manual Rollback**: Performance issues, user complaints
3. **Emergency Rollback**: Critical system failures

### 10.2 Rollback Procedure

Emergency rollback procedures involve stopping the Python service and starting the OCaml fallback service on the production VM. The system verifies the rollback success by checking service status and updates load balancer health checks to route traffic to the OCaml implementation. Alternative rollback methods include direct Docker container management for immediate service switching.

## Success Criteria

### Technical Success Metrics

- ☐ 100% Golden Master test pass rate
- ☐ Performance parity or improvement vs OCaml
- ☐ Zero-downtime deployments
- ☐ $< 1\%$ error rate in production
- ☐ Successful fallback mechanism validation

### Business Success Metrics

- ☐ Reduced maintenance overhead
- ☐ Improved development velocity
- ☐ Enhanced system observability
- ☐ Cost optimization achieved
- ☐ Team knowledge transfer completed

## Team Responsibilities

### Development Team

- Python implementation development
- Test suite maintenance
- Code review and quality assurance
- Performance optimization

### DevOps Team

- CI/CD pipeline management
- Infrastructure provisioning
- Monitoring and alerting setup
- Deployment automation

### QA Team

- Test strategy development
- Golden Master test validation
- Performance testing

- User acceptance testing

## Risk Mitigation

### High-Risk Scenarios

1. **Python Performance Issues**: OCaml fallback ready
2. **Data Corruption**: Comprehensive backup strategy
3. **Infrastructure Failures**: Multi-region deployment
4. **Team Knowledge Gap**: Documentation and training

### Mitigation Strategies

- **Comprehensive Testing**: Multiple test layers
- **Gradual Migration**: Phased approach with validation
- **Monitoring**: Real-time system health tracking
- **Rollback Plans**: Tested and automated procedures

---

*This deployment strategy ensures a safe, reliable, and efficient migration from OCaml to Python while maintaining system stability and user experience.*