Knative Serverless Platform Hands-on Lab Report

1. Introduction and Objectives

This comprehensive lab exercise provides practical experience with Knative, a Kubernetes-based platform for building, deploying, and managing serverless, cloud-native applications. The experiment focuses on three core Knative components: Knative Functions for function-based development, Knative Serving for service deployment and autoscaling, and Knative Eventing for event-driven architectures.

The primary objectives are to:

- Understand Knative's serverless application model and its integration with Kubernetes
- Explore function-based development using Knative Functions
- Deploy and manage scalable services using Knative Serving
- Implement event-driven architectures using Knative Eventing
- Gain hands-on experience with local development workflows using container registries

2. Environment Setup and Prerequisites

2.1 Infrastructure Preparation

The experiment begins with establishing a local Kubernetes development environment using kind (Kubernetes in Docker). A custom cluster configuration is created to support Knative's networking requirements,

including port mappings for ingress controllers and proper node labeling for traffic management.

The local development environment includes:

- Container runtime for building and storing application images
- Local container registry for development workflow
- Kubernetes cluster with appropriate resource allocation
- Command-line tools for cluster and application management

2.2 Knative Platform Installation

The installation process involves deploying three main Knative components in sequence. First, Istio service mesh is installed as the networking layer, providing traffic management capabilities required by Knative Serving. The installation uses the demo configuration profile, which includes all necessary components for development and testing.

Next, Knative Serving components are deployed, including the core controller, webhook, and automatic scaling capabilities. The installation also includes Istio integration for traffic routing and a default domain configuration for local development. Finally, Knative Eventing components are installed to support event-driven application patterns.

2.3 Verification and Validation

After installation, the system undergoes comprehensive validation to ensure all components are running correctly. This includes verifying pod status across

all Knative namespaces, checking service availability, and validating that the control plane components are operational.

3. Knative Functions Experience

3.1 Function Development Environment

Knative Functions provides a simplified programming model for creating serverless functions without requiring deep knowledge of containers or Kubernetes. The experiment explores this capability through the func CLI tool, which enables function creation, building, and local development.

The function development workflow begins with creating a new function project. The tool generates a complete project structure including function code, configuration files, and build specifications. The generated function is a simple HTTP handler that can be customized for specific use cases.

3.2 Local Development Workflow

The local development experience includes the ability to run functions locally for testing and debugging. This capability allows developers to validate function behavior before deployment, reducing development cycle time and improving code quality.

The local registry setup enables complete offline development workflow. The container registry runs locally, allowing function images to be built and stored without requiring external connectivity. This approach demonstrates how

Knative Functions abstracts the complexity of container image management while maintaining full compatibility with container standards.

3.3 Build Process Understanding

The function build process automatically generates OCI-compliant container images from function code. This process demonstrates how Knative Functions eliminates the need for Docker knowledge while still producing standard container images that can run anywhere.

The build process includes dependency resolution, security scanning, and image optimization. These capabilities ensure that functions are ready for production deployment while maintaining the simplicity of the development experience.

4. Knative Serving Deep Dive

4.1 Service Deployment Experience

Knative Serving transforms Kubernetes into a serverless application platform by providing automatic scaling, traffic routing, and revision management. The experiment demonstrates these capabilities through deploying a simple "Hello World" service.

The deployment process reveals several key capabilities:

- Automatic container image building and deployment
- Instant service availability with URL assignment
- Automatic TLS certificate generation for secure access

- Built-in traffic management and load balancing

4.2 Autoscaling Demonstration

One of Knative Serving's most powerful features is automatic scaling based on incoming traffic. The experiment demonstrates this capability by monitoring how the platform responds to varying load levels.

Key observations include:

- **Scale-to-Zero**: When no traffic is received, the service automatically scales down to zero instances, conserving resources
- **Rapid Scale-Up**: When traffic arrives, the platform quickly creates new instances to handle the load
- **Graceful Scale-Down**: After traffic subsides, instances are gradually terminated to maintain responsiveness while optimizing resource usage
- **Concurrency Management**: The platform intelligently manages request distribution across instances

4.3 Service Update and Revision Management

The experiment explores how Knative Serving handles service updates through its revision management system. Each service update creates a new revision, enabling sophisticated deployment strategies including blue-green deployments, canary releases, and instant rollbacks.

The revision system provides:

- **Traffic Splitting**: Ability to route traffic between multiple revisions
- **Rollback Capability**: Instant reversion to previous revisions if issues arise
- **Progressive Deployment**: Gradual traffic shifting to new versions
- **Configuration Management**: Complete configuration history and audit trail

4.4 Traffic Management and Routing

Knative Serving includes advanced traffic management capabilities that enable sophisticated routing patterns. The platform supports header-based routing, percentage-based traffic splitting, and A/B testing scenarios.

These capabilities enable development teams to:

- Test new features with specific user segments
- Gradually roll out changes while monitoring for issues
- Implement feature flags and toggles
- Perform comprehensive testing in production environments

5. Knative Eventing Implementation

5.1 Event-Driven Architecture Concepts

Knative Eventing provides building blocks for creating event-driven applications on Kubernetes. The component enables loose coupling between event producers and consumers, supporting various messaging patterns including pub/sub, event sourcing, and CQRS.

The experiment explores fundamental eventing concepts including:

- **Event Sources**: Components that generate events from external systems
- **Brokers**: Event distribution hubs that manage event delivery
- **Triggers**: Rules that determine which events should be delivered to which services
- **Sinks**: Services that receive and process events

5.2 Broker and Trigger Configuration

The experiment demonstrates setting up an eventing infrastructure with a default broker and trigger configuration. The broker acts as a central hub for

event distribution, while triggers define routing rules based on event attributes.

This configuration enables:

- **Decoupled Architecture**: Event producers don't need to know about event consumers
- **Flexible Routing**: Events can be routed based on type, source, or custom attributes
- **Multiple Consumers**: Single events can trigger multiple services
- **Reliable Delivery**: Built-in retry and dead letter queue capabilities

5.3 Event Source Integration

The experiment includes creating event sources that generate test events. These sources demonstrate how external systems can integrate with Knative Eventing to trigger application logic.

Event source capabilities include:

- **CloudEvents Format**: Standardized event format for interoperability
- **Batch Processing**: Ability to handle high-volume event streams
- **Filtering and Transformation**: Event preprocessing before delivery
- **Reliability Guarantees**: At-least-once delivery semantics

5.4 Event Processing and Response

The event processing demonstration shows how services can receive and respond to events. The experiment includes creating an event display service that logs received events, providing visibility into the eventing system operation.

Key aspects of event processing:

- **Automatic Scaling**: Services scale based on event volume
- **Event Acknowledgment**: Built-in acknowledgment mechanisms

- **Error Handling**: Comprehensive error handling and retry logic
- **Observability**: Detailed metrics and tracing for event flows

6. Integration and Workflow Analysis

6.1 End-to-End Serverless Workflow

The experiment demonstrates how Knative components work together to provide a complete serverless platform. Functions can be triggered by events, services can scale automatically based on demand, and the entire system operates without manual intervention.

This integration enables:

- **Event-Driven Functions**: Functions that respond to events automatically
- **Service Chaining**: Events that trigger multiple services in sequence
- **Stateful Processing**: Support for stateful event processing patterns
- **Complex Workflows**: Multi-step business processes orchestrated through events

6.2 Development Experience Analysis

The overall development experience with Knative demonstrates significant improvements in developer productivity through:

- **Reduced Complexity**: Abstracted infrastructure management
- **Faster Development Cycles**: Local development and testing capabilities
- **Standardized Deployment**: Consistent deployment patterns across applications
- **Built-in Best Practices**: Automatic inclusion of scalability, security, and observability

6.3 Operational Benefits

From an operational perspective, Knative provides several advantages:

- **Resource Optimization**: Automatic scaling prevents over-provisioning

- **High Availability**: Built-in health checking and self-healing capabilities
- **Security**: Automatic TLS and security policy enforcement
- **Observability**: Comprehensive metrics, logging, and tracing

7. Local Development Workflow Insights

7.1 Local Registry Benefits

Using a local container registry for development provides several advantages:

- **Offline Development**: Complete independence from external networks
- **Faster Iteration**: Local image storage and retrieval
- **Cost Efficiency**: No external registry costs during development
- **Security**: Sensitive code never leaves the development environment

7.2 Development Cycle Optimization

The experiment demonstrates how Knative optimizes the development cycle through:

- **Hot Reloading**: Quick iteration without full rebuilds
- **Local Testing**: Complete local testing environment
- **Production Parity**: Local environment closely matches production
- **Rapid Feedback**: Immediate validation of changes

7.3 Debugging and Troubleshooting

Knative provides comprehensive debugging capabilities:

- **Detailed Logging**: Extensive logging throughout the platform
- **Metrics Collection**: Built-in metrics for performance analysis
- **Distributed Tracing**: Request tracing across service boundaries
- **Health Checking**: Automatic health checking and reporting

8. Performance and Scalability Observations

8.1 Cold Start Performance

The experiment reveals insights into cold start performance, including:

- **Startup Time**: Time required to scale from zero to handling requests
- **Optimization Strategies**: Techniques for minimizing cold start impact
- **Resource Pre-warming**: Strategies for maintaining optimal performance
- **Concurrency Handling**: How the platform manages concurrent requests during scaling

8.2 Scaling Behavior Analysis

Detailed analysis of scaling behavior includes:

- **Scale-Up Speed**: How quickly the platform responds to increased load
- **Scale-Down Efficiency**: How effectively resources are reclaimed during low usage
- **Request Queuing**: How the platform handles requests during scaling operations
- **Resource Utilization**: Efficiency of resource allocation and usage

8.3 Resource Efficiency

The platform demonstrates several resource efficiency optimizations:

- **Bin Packing**: Efficient placement of workloads on available resources
- **Right-Sizing**: Automatic adjustment of resource allocations based on usage
- **Idle Resource Management**: Aggressive reclamation of unused resources
- **Multi-tenancy**: Efficient sharing of infrastructure across multiple applications

9. Security and Compliance Considerations

9.1 Built-in Security Features

Knative includes several security features that are automatically applied:

- **TLS Everywhere**: Automatic TLS certificate generation and management
- **Network Policies**: Fine-grained network access controls

- **Service-to-Service Security**: Mutual TLS for service communication
- **Identity and Access Management**: Integration with Kubernetes RBAC

9.2 Compliance Implications

The platform supports various compliance requirements through:

- **Audit Logging**: Comprehensive audit trails for all operations
- **Data Isolation**: Strong isolation between different applications and tenants
- **Policy Enforcement**: Automated enforcement of security policies
- **Vulnerability Management**: Built-in scanning and remediation capabilities

10. Cost Optimization Analysis

10.1 Resource Cost Benefits

The experiment demonstrates several cost optimization benefits:

- **Pay-per-Use Pricing**: Only pay for resources actually consumed
- **Automatic Resource Optimization**: No over-provisioning required
- **Idle Resource Elimination**: Zero cost during idle periods
- **Efficient Resource Sharing**: Multi-tenant efficiency improvements

10.2 Operational Cost Reduction

Operational cost benefits include:

- **Reduced Management Overhead**: Automated infrastructure management
- **Faster Time-to-Market**: Reduced development and deployment time
- **Lower Skill Requirements**: Reduced need for specialized infrastructure knowledge
- **Improved Reliability**: Reduced downtime and associated costs

11. Integration with Existing Systems

11.1 Kubernetes Compatibility

The experiment demonstrates full compatibility with standard Kubernetes tools and processes:

- **kubectl Integration**: Complete functionality through standard kubectl commands
- **Helm Support**: Full support for Helm charts and templating
- **GitOps Integration**: Compatibility with GitOps workflows and tools
- **Monitoring Integration**: Works with standard Kubernetes monitoring solutions

11.2 CI/CD Pipeline Integration

Knative integrates seamlessly with existing CI/CD pipelines:

- **Automated Deployment**: Easy integration with deployment automation
- **Progressive Delivery**: Support for advanced deployment strategies
- **Rollback Capabilities**: Instant rollback when issues are detected
- **Environment Promotion**: Easy promotion between environments

12. Limitations and Considerations

12.1 Platform Limitations

The experiment reveals several platform limitations:

- **Cold Start Latency**: Inherent latency when scaling from zero
- **Stateless Focus**: Primarily designed for stateless applications
- **Resource Constraints**: Minimum resource requirements for operation
- **Complexity for Simple Cases**: May be overkill for very simple applications

12.2 Operational Considerations

Operational considerations include:

- **Learning Curve**: Requires understanding of Kubernetes and serverless concepts
- **Debugging Complexity**: Distributed nature can complicate debugging

- **Vendor Lock-in**: Potential lock-in to specific serverless platforms
- **Cost Predictability**: Pay-per-use model can make costs less predictable

13. Future Development Opportunities

13.1 Advanced Use Cases

The experiment establishes a foundation for exploring more advanced use cases:

- **Machine Learning Integration**: Serving ML models with automatic scaling
- **IoT Data Processing**: Processing high-volume IoT data streams
- **Real-time Analytics**: Real-time processing and analysis of data streams
- **Multi-cloud Deployments**: Deploying across multiple cloud providers

13.2 Platform Extensions

Future platform developments could include:

- **Enhanced State Management**: Better support for stateful applications
- **Improved Cold Start Performance**: Reduced latency for function startup
- **Advanced AI/ML Integration**: Built-in support for AI/ML workloads
- **Edge Computing Support**: Extension to edge computing scenarios

14. Conclusions and Key Insights

14.1 Technical Achievements

This comprehensive experiment successfully demonstrates Knative's capabilities as a serverless platform for Kubernetes. The platform effectively abstracts infrastructure complexity while providing powerful capabilities for building, deploying, and managing serverless applications.

Key technical achievements include:

- Successful deployment and operation of all three Knative components
- Demonstration of automatic scaling from zero to handling load
- Implementation of event-driven architectures with loose coupling
- Validation of local development workflows with container registries

14.2 Business Value

The experiment reveals significant business value through:

- **Reduced Operational Complexity**: Automated infrastructure management reduces operational overhead
- **Improved Developer Productivity**: Simplified development and deployment processes
- **Enhanced Scalability**: Automatic scaling ensures applications can handle varying loads
- **Cost Optimization**: Pay-per-use model reduces infrastructure costs

14.3 Strategic Implications

The strategic implications of adopting Knative include:

- **Cloud-Native Transformation**: Enables organizations to adopt cloud-native practices
- **Developer Empowerment**: Allows developers to focus on business logic rather than infrastructure
- **Operational Excellence**: Provides enterprise-grade operational capabilities
- **Future-Proofing**: Positions organizations for future technology developments

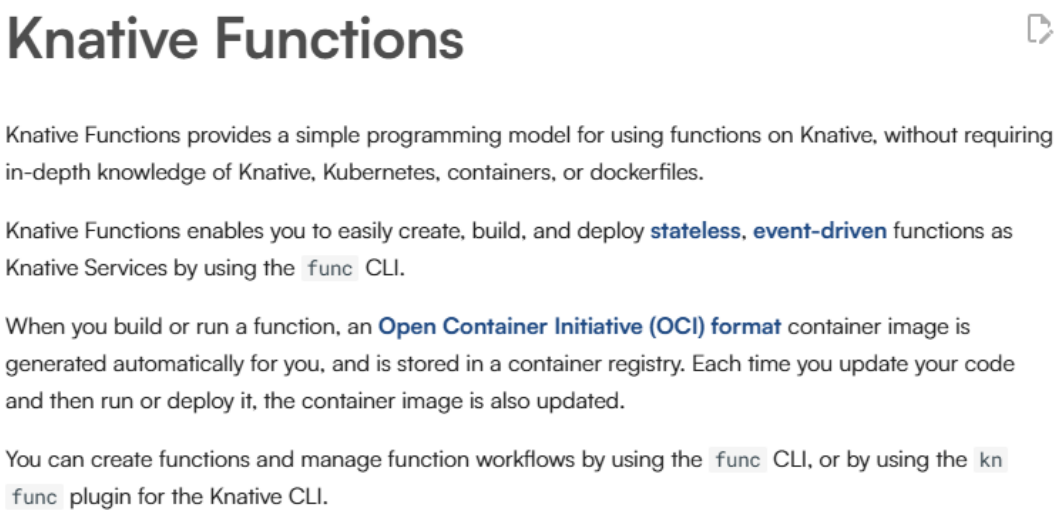
14.4 Recommendations

Based on the experimental results, recommendations for organizations considering Knative include:

- Start with simple use cases to build familiarity with the platform

- Invest in training and skill development for development and operations teams
- Establish clear governance and best practices for function and service development
- Implement comprehensive monitoring and observability from the beginning
- Plan for gradual adoption rather than immediate full-scale deployment

This experiment provides a solid foundation for understanding Knative's serverless capabilities and demonstrates how the platform can enable organizations to build more scalable, efficient, and manageable cloud-native applications.



Knative Functions

Knative Functions provides a simple programming model for using functions on Knative, without requiring in-depth knowledge of Knative, Kubernetes, containers, or dockerfiles.

Knative Functions enables you to easily create, build, and deploy **stateless**, **event-driven** functions as Knative Services by using the `func` CLI.

When you build or run a function, an **Open Container Initiative (OCI) format** container image is generated automatically for you, and is stored in a container registry. Each time you update your code and then run or deploy it, the container image is also updated.

You can create functions and manage function workflows by using the `func` CLI, or by using the `kn func` plugin for the Knative CLI.

```
PS C:\WINDOWS\system32> Set-ExecutionPolicy Bypass -Scope Process -Force
>> [System.Net.ServicePointManager]::SecurityProtocol = [System.Net.ServicePointManager]::SecurityProtocol -bor 3072
>> iex ((New-Object System.Net.WebClient).DownloadString('https://community.chocolatey.org/install.ps1'))
Forcing web requests to allow TLS v1.2 (Required for requests to Chocolatey.org)
Getting latest version of the Chocolatey package for download.
Not using proxy.
Getting Chocolatey from https://community.chocolatey.org/api/v2/package/chocolatey/2.6.0.
Downloading https://community.chocolatey.org/api/v2/package/chocolatey/2.6.0 to C:\Users\25862\AppData\Local\Temp\chocolatey\chocoInstall\chocolatey.zip
Not using proxy.
Extracting C:\Users\25862\AppData\Local\Temp\chocolatey\chocoInstall\chocolatey.zip to C:\Users\25862\AppData\Local\Temp\chocolatey\chocoInstall
Installing Chocolatey on the local machine
Creating ChocolateyInstall as an environment variable (targeting 'Machine')
  Setting ChocolateyInstall to 'C:\ProgramData\chocolatey'
WARNING: It's very likely you will need to close and reopen your shell
  before you can use choco.
Restricting write permissions to Administrators
We are setting up the Chocolatey package repository.
The packages themselves go to 'C:\ProgramData\chocolatey\lib'
  (i.e. C:\ProgramData\chocolatey\lib\yourPackageName).
A shim file for the command line goes to 'C:\ProgramData\chocolatey\bin'
  and points to an executable in 'C:\ProgramData\chocolatey\lib\yourPackageName'.

Creating Chocolatey CLI folders if they do not already exist.

chocolatey.nupkg file not installed in lib.
 Attempting to locate it from bootstrapper.
PATH environment variable does not have C:\ProgramData\chocolatey\bin in it. Adding...
警告: Not setting tab completion: Profile file does not exist at 'C:\Users\25862\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1'.
Chocolatey CLI (choco.exe) is now ready.
You can call choco from anywhere, command line or PowerShell by typing choco.
Run choco /? for a list of functions.
You may need to shut down and restart PowerShell and/or consoles
 first prior to using choco.
Ensuring Chocolatey commands are on the path
Ensuring chocolatey.nupkg is in the lib folder
PS C:\WINDOWS\system32>
PS C:\WINDOWS\system32> choco --version
2.6.0
```

```
Chocolatey-dotnetfx.extension package files install completed. Performing other installation steps.
 Installed/updated chocolatey-dotnetfx extensions.
 The install of chocolatey-dotnetfx.extension was successful.
  Deployed to 'C:\ProgramData\chocolatey\extensions\chocolatey-dotnetfx'
Downloading package from source 'https://community.chocolatey.org/api/v2/'
Progress: Downloading KB2919442 1.0.20160915... 100%

KB2919442 v1.0.20160915 [Approved]
KB2919442 package files install completed. Performing other installation steps.
Skipping installation because this hotfix only applies to Windows 8.1 and Windows Server 2012 R2.
 The install of KB2919442 was successful.
  Software install location not explicitly set, it could be in package or
  default install location of installer.
Downloading package from source 'https://community.chocolatey.org/api/v2/'
Progress: Downloading KB2919355 1.0.20160915... 100%

KB2919355 v1.0.20160915 [Approved]
KB2919355 package files install completed. Performing other installation steps.
Skipping installation because this hotfix only applies to Windows 8.1 and Windows Server 2012 R2.
 The install of KB2919355 was successful.
  Software install location not explicitly set, it could be in package or
  default install location of installer.
Downloading package from source 'https://community.chocolatey.org/api/v2/'
Progress: Downloading dotnetfx 4.8.0.20220524... 100%

dotnetfx v4.8.0.20220524 [Approved]
dotnetfx package files install completed. Performing other installation steps.
Microsoft .NET Framework 4.8 or later is already installed.
 The install of dotnetfx was successful.
  Software install location not explicitly set, it could be in package or
  default install location of installer.
Downloading package from source 'https://community.chocolatey.org/api/v2/'
Progress: Downloading docker-desktop 4.54.0... 100%

docker-desktop v4.54.0 [Approved]
docker-desktop package files install completed. Performing other installation steps.
Using system proxy server '127.0.0.1:7890'.
Downloading docker-desktop 64 bit
  from 'https://desktop.docker.com/win/main/amd64/212467/Docker%20Desktop%20Installer.exe'
Using system proxy server '127.0.0.1:7890'.
Progress: 65% - Saving 373.62 MB of 573.03 MB
```