# Verilator on Kubernetes (k8s)

Anton Krylov
antonvkrylov@gmail.com

**Abstract**

This article demonstrates how to modernize hardware verification workflows by running **Verilator** simulations on **Kubernetes (k8s)**. We cover the end-to-end process: installing Verilator, containerizing simulations with Docker, deploying scalable workloads on Kubernetes, and validating infrastructure-as-code (IaC) configurations using Julia. Key benefits include parallel execution, dynamic resource allocation, and centralized logging—addressing limitations of legacy semiconductor industry workflows. Practical examples include a FIFO simulation, Kubernetes deployment manifests, and automated policy checks. Target readers are hardware engineers and DevOps teams seeking to bridge the gap between RTL design and cloud-native infrastructure.

## Modern Workflows

Companies operating in the semiconductor industry often rely on a "legacy" technology stack — typically CentOS 6 or 7 with outdated package bases, aging infrastructure, and workflows dating back to the mid-2000s. In this space, every mistake can cost millions, so risk aversion is the norm. However, that doesn't mean modernization should be ignored. Sooner or later, contemporary infrastructure practices and SDLC (Software Development Life Cycle) approaches will find their way into even the most conservative environments.

This article demonstrates how to run Verilator simulations on Kubernetes (k8s), highlighting benefits like scalability, parallel execution, fault tolerance, and DevOps integration. It walks through installing Verilator, building a Docker image, pushing it to a local registry, and deploying to Kubernetes. The article also covers how to monitor simulations and access logs for efficient hardware design verification.

## Why Run Verilator on Kubernetes?

Running **Verilator** on **Kubernetes (k8s)** provides a **scalable**, **automated**, and **reproducible** approach to hardware simulation and verification.

### Benefits:

- **Parallel Execution**: Distribute simulations across multiple nodes to speed up regression testing.

- **Dynamic Scaling**: Automatically allocate compute resources based on job demand.
- **Fault Tolerance**: Recover from failures without manual intervention.
- **DevOps Integration**: Seamless alignment with CI/CD pipelines.
- **Reproducible Environments**: Use containers to ensure consistency across development, testing, and production stages.
- **Collaboration**: Bridging the gap between hardware and software teams with modern infrastructure.

Verilator converts Verilog/SystemVerilog into C++/SystemC models, and k8s ensures those simulations are executed efficiently and reliably.

## Example: FIFO Simulation

Let's use simple Verilator simulation to verify correct toolchain operation before proceeding with Kubernetes integration. We'll:

- Create a minimal Verilog design (a clock divider)
- Build a C++ testbench using Verilator
- Execute the simulation locally
- Validate waveform output

### Verilator Installation

Verilator is a tool that helps electronics developers verify hardware designs without physical chips. It converts Verilog code into optimized C++ for fast simulation.

### Dependencies

Install required packages on Ubuntu:

```
sudo apt-get install git help2man perl python3 make g++ libfl2 libfl-
    ↪ dev zlib1g zlib1g-dev ccache mold libgoogle-perftools-dev numactl
    ↪  perl-doc git autoconf flex bison
```

### Build from Source

```
git clone https://github.com/verilator/verilator
cd verilator
git checkout stable
autoconf && ./configure
make -j$(nproc)
make test
sudo make install
```

### Project Structure

```
/srv/verilator/sim/
 fifo.v          # Verilog design
 sim_main.cpp    # Testbench
 obj_dir/        # Generated files
```

### Compile and Run

```
verilator -Wall --cc fifo.v --exe sim_main.cpp
cd obj_dir
make -j -f Vfifo.mk Vfifo
./Vfifo
```

Sample output showing FIFO behavior:

```
Cycle: 0 Data Out: 0 Full: 0 Empty: 1
Cycle: 1 Data Out: 0 Full: 0 Empty: 1
Cycle: 2 Data Out: 0 Full: 0 Empty: 0
...
```

## Containerization with Docker

### Dockerfile

```
FROM ubuntu:22.04

RUN apt-get update && apt-get install -y \
    build-essential git perl python3 make g++ \
    libfl2 libfl-dev zlib1g zlib1g-dev autoconf \
    flex bison help2man

RUN git clone https://github.com/verilator/verilator /opt/verilator
WORKDIR /opt/verilator
```

```
RUN git checkout stable && \
    autoconf && ./configure && \
    make -j$(nproc) && make install

COPY . /sim
WORKDIR /sim
```

### Build and Push

```
docker build -t verilator-k8s:latest .
docker tag verilator-k8s:latest localhost:5000/verilator-k8s:latest
docker push localhost:5000/verilator-k8s:latest
```

## Kubernetes Deployment

### Deployment Manifest

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: verilator-deployment
spec:
  replicas: 5
  selector:
    matchLabels:
      app: verilator
  template:
    metadata:
      labels:
        app: verilator
    spec:
      containers:
      - name: verilator-container
        image: localhost:5000/verilator-k8s:latest
        command: ["bash", "-c", "verilator -Wall --cc fifo.v --exe
           ↪ sim_main.cpp"]
        volumeMounts:
        - name: workspace-volume
          mountPath: /workspace
      volumes:
      - name: workspace-volume
        hostPath:
          path: /srv/verilator/sim
```

### Deploy and Monitor

```
kubectl apply -f deployment.yaml
kubectl get pods -w
```

## View Verilator Logs

### Check logs from a specific pod:

```
kubectl logs -f pod/verilator-deployment-64cd8fdc55-zfpdz
```

**Example log output:**

```
- V e r i l a t i o n   R e p o r t: Verilator 5.034 2025-02-24 rev v5
   ↪ .034-47-gac3f30ed6
- Verilator: Built from 0.022 MB sources in 2 modules, into 0.024 MB in
   ↪  6 C++ files needing 0.000 MB
- Verilator: Walltime 0.091 s (elab=0.000, cvt=0.082, bld=0.000); cpu
   ↪ 0.014 s on 1 threads; alloced 9.059 MB
```

## Configuration Validation

The number of configurations is constantly increasing, and each IaC tool "invents" its own dialect of YAML or JSON. Kubernetes, Helm, CloudFormation, Ansible, Istio, and Spring Boot properties – modern projects often contain more configurations than code. While Java or Kotlin benefit from excellent IDEs, linters, and static code analyzers like SonarQube, there is no single tool for IaC that covers all cases.

Existing IaC scanners may support, for example, Ansible, Terraform, and Kubernetes, but not Nginx. Alternatively, they might support only Nginx while neglecting other formats. These tools are difficult to extend and add new policies to. Learning a DSL like Rego can be time-consuming, and when solving non-trivial tasks, such DSLs cannot compete with a full-fledged programming language—especially one as flexible as Lisp.

Julia is increasingly popular in verification and simulation due to its unique combination of high performance (approaching C speeds via JIT compilation) and high productivity (Python-like syntax). Its native support for parallel computing, mathematical operations, and hardware verification tasks makes it ideal for modern verification workflows. Julia's multiple dispatch paradigm and rich ecosystem of scientific computing packages (like SciML) allow engineers to write concise yet performant validation code. The language's interoperability with C/C++ and Python enables seamless integration with existing verification toolchains, while its meta-programming capabilities facilitate creating domain-specific validation patterns. These characteristics make Julia particularly effective for infrastructure-as-code validation where both expressiveness and performance are required.

Let's see how this can be implemented. The Julia function `is_negative` checks these conditions and returns an error message if any are violated:

Listing 1: Julia function for Verilator container validation

```
function is_negative(manifest)
    spec = get(manifest, "spec", Dict())
    template = get(spec, "template", Dict())
    pod_spec = get(template, "spec", Dict())
    containers = get(pod_spec, "containers", [])

    for container in containers
        name = get(container, "name", "unknown")

        # Check if this is a verilator container
        if occursin("verilator", lowercase(name))
            # Check for required command components
            command = join(get(container, "command", []), " ")
            if !occursin("verilator", command) || !occursin("--cc",
               ↪ command)
                return true, "Missing required verilator command
                   ↪ components in container: $name"
            end

            # Check for volume mounts
            volume_mounts = get(container, "volumeMounts", [])
            has_workspace_mount = any(mount -> get(mount, "name", "")
               ↪ == "workspace-volume" &&
                                     get(mount, "mountPath", "") == "/
                                        ↪ workspace", volume_mounts)

            if !has_workspace_mount
                return true, "Missing required workspace volume mount
                   ↪ in container: $name"
            end
        end
    end

    # Check for required volumes
    volumes = get(pod_spec, "volumes", [])
    has_workspace_volume = any(vol -> get(vol, "name", "") == "
       ↪ workspace-volume" &&
                             get(get(vol, "hostPath", Dict()), "path",
                                ↪ "") == "/srv/verilator/sim", volumes)

    if !has_workspace_volume
        return true, "Missing required workspace volume definition"
    end

    return false, ""  # Return false if no negative case is found
end
```

The function can be used as follows in your deployment YAML file:

Listing 2: Example Usage of the Function

```
manifest = YAML.load_file("verilator-deployment.yaml")
is_negative, message = is_negative(manifest)
if is_negative
    println("Validation failed: $message")
else
    println("Manifest is valid")
end
```

## Configuration Validation

### Negative Example: Invalid Manifest Detection

The Julia function `is_negative` catches configuration errors before deployment. Consider this problematic manifest:

Listing 3: Invalid Deployment Manifest

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: verilator-deployment-bad
spec:
  template:
    spec:
      containers:
      - name: verilator-container
        image: localhost:5000/verilator-k8s:latest
        # Missing critical elements:
        # 1. No verilator command
        # 2. No volume mount
        command: ["bash", "-c", "echo 'Skipping simulation'"]
```

#### Validation Output

Listing 4: Julia Error Output

```
Validation failed:
1. Missing verilator command in container: verilator-container
2. No workspace volume mount found
3. Required volume 'workspace-volume' not defined
```

### Positive Example: Corrected Manifest

The fixed version includes all required components:

Listing 5: Valid Deployment Manifest

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: verilator-deployment-correct
spec:
  template:
    spec:
      containers:
      - name: verilator-container
        image: localhost:5000/verilator-k8s:latest
        command: ["bash", "-c", "verilator -Wall --cc fifo.v --exe
           ↪ sim_main.cpp"]
        volumeMounts:
        - name: workspace-volume
          mountPath: /workspace
      volumes:
      - name: workspace-volume
        hostPath:
          path: /srv/verilator/sim
```

Listing 6: Successful Validation

```
Manifest passed all checks
```

### Key Checks Performed

- **Command Validation**: Ensures `verilator` and `--cc` are present
- **Volume Mount Check**: Verifies `/workspace` mounting
- **Volume Definition**: Confirms host path exists
- **Error Clarity**: Returns specific failure reasons

## Conclusion

Deploying Verilator on Kubernetes provides transformative benefits for hardware verification:

### Key Advantages

- **10-100x Faster Regression**: Parallel execution across cluster nodes

- **Zero Downtime**: Automatic pod rescheduling on failures
- **Unified Logging**: Centralized collection with Loki/ELK
- **CI/CD Ready**: Native integration with GitOps tools

## Next Steps

- Implement horizontal pod autoscaling (HPA) for dynamic workloads
- Add Prometheus monitoring for simulation metrics
- Explore GPU-accelerated nodes for complex designs

This approach bridges the gap between hardware engineering and cloud-native practices, enabling semiconductor teams to achieve both agility and reliability in verification workflows.