Watchtower — Local LGTM Observability Stack in Kubernetes

Project Name

```
watchtower
```

Overview

Watchtower is a local Kubernetes-hosted observability platform running the full Grafana LGTM stack (Loki, Grafana, Tempo, Mimir) with Grafana Alloy as the telemetry collector and router. It runs on a work MacBook Pro using **kind** (Kubernetes IN Docker) and serves two purposes:

1. **POC for Grafana LGTM** — evaluate whether this stack can replace or augment existing Sumo Logic and Datadog observability at work
2. **Learn Kubernetes** — hands-on experience with K8s primitives, Helm, pod networking, resource management, and observability patterns in a realistic but safe local environment

The key architectural pattern is **Alloy as a telemetry router**: Alloy receives OTLP from instrumented applications and dual-writes to both the local LGTM stack (for low-latency experimentation) and Sumo Logic (preserving existing production observability). This lets the team run experimental analysis, build dashboards, and evaluate Grafana tooling against real telemetry without disrupting the production Sumo pipeline.

Why kind

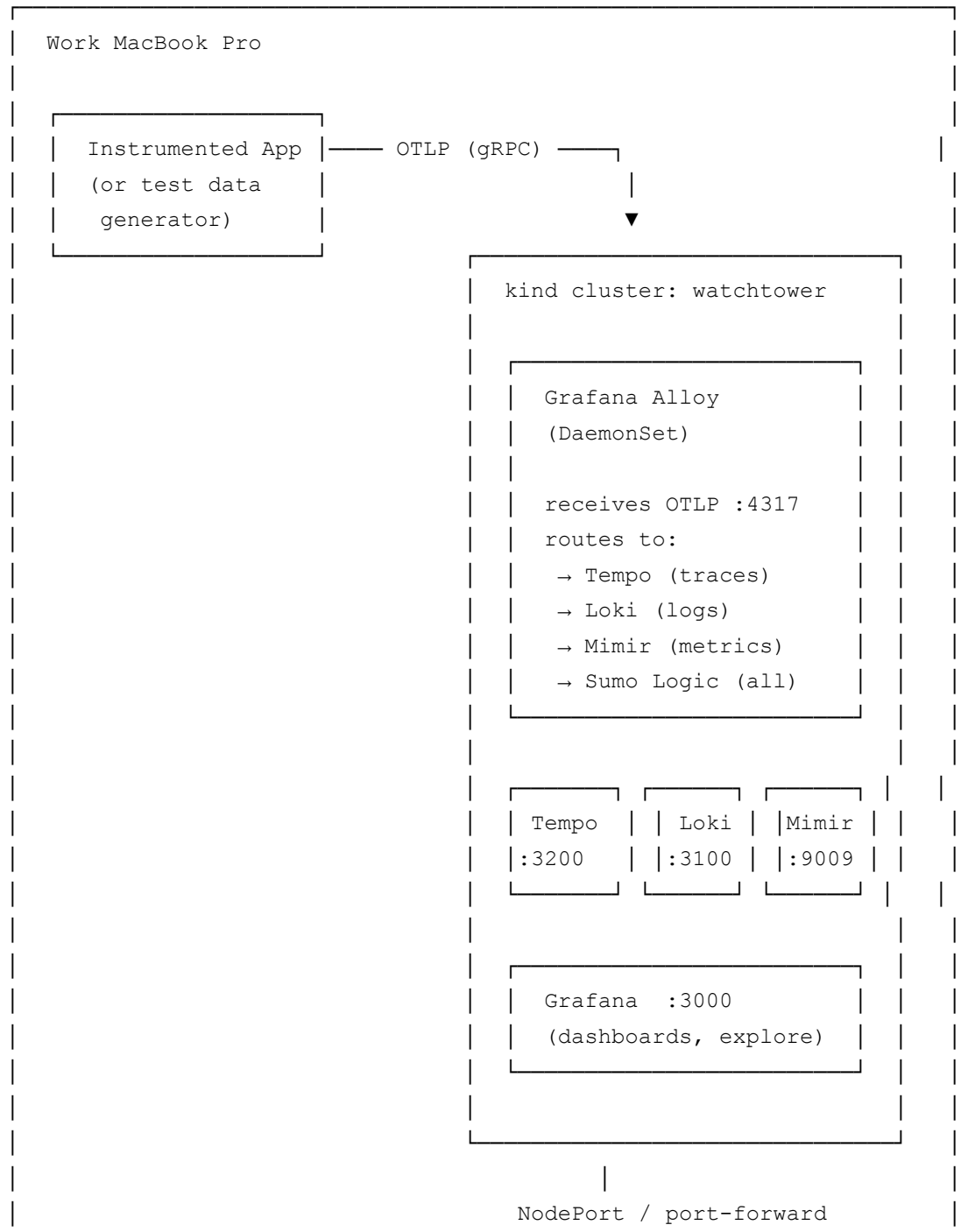
kind (Kubernetes IN Docker) runs real K8s nodes as Docker containers. It uses containerd, the same container runtime as EKS, and exposes the standard K8s API. The gap between kind and EKS is mostly networking (CNI, load balancers) and IAM (IRSA, service accounts) — not the K8s primitives themselves. This makes it the best local option for building skills that transfer to AWS.

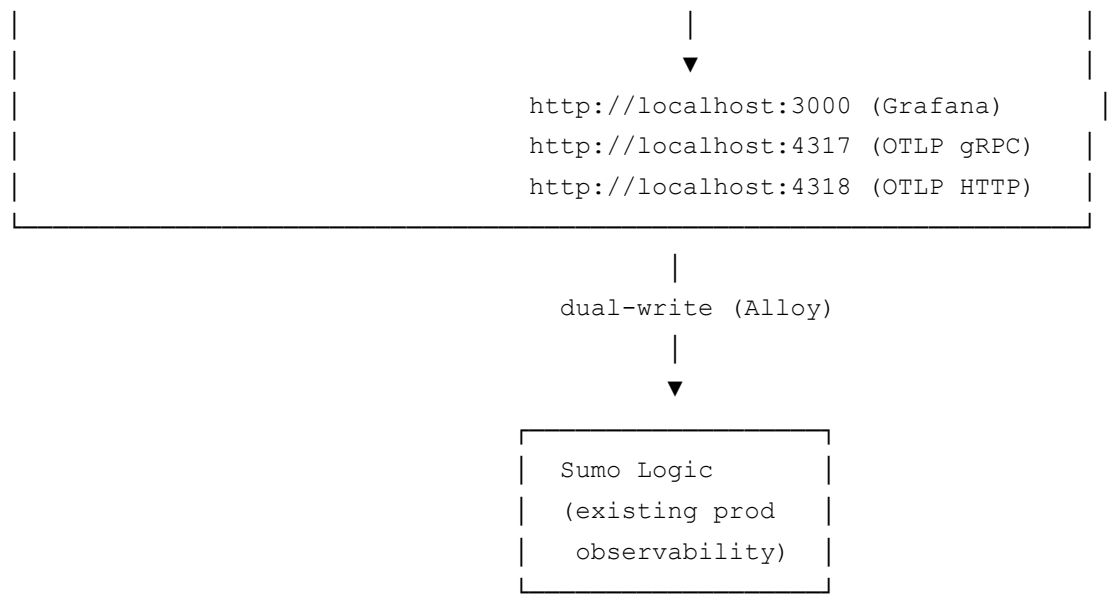
Requirements:

- Docker Desktop installed on the MacBook Pro

- `kind` CLI (`brew install kind`)
- `kubectl` CLI (`brew install kubectl`)
- `helm` CLI (`brew install helm`)
- Sufficient RAM allocated to Docker Desktop (recommend 8GB minimum, 12GB preferred)

Architecture





Repository Structure

```
watchtower/
├── README.md
├── Makefile                          # Common operations (see below)
├── kind/
│   └── cluster-config.yaml           # kind cluster definition
├── helm/
│   ├── values/
│   │   ├── grafana.yaml              # Grafana Helm values
│   │   ├── tempo.yaml                # Tempo Helm values
│   │   ├── loki.yaml                 # Loki Helm values
│   │   ├── mimir.yaml                # Mimir Helm values
│   │   └── alloy.yaml                # Alloy Helm values
│   └── rendered/                     # Optional: `helm template` output for lear
│       └── .gitkeep
├── alloy/
│   ├── config.alloy                  # Primary Alloy config (dual-write: LGTM +
│   └── config-local-only.alloy       # Local LGTM only (fallback if Sumo is unav
├── dashboards/
│   └── .gitkeep                      # Provisioned Grafana dashboards (JSON)
├── test-data/
│   ├── generate.py                   # Synthetic telemetry generator
│   ├── requirements.txt              # Python deps for generator
│   └── README.md                     # How to run the generator
├── scripts/
│   ├── setup.sh                      # One-time setup (install tools, create clu
│   ├── teardown.sh                   # Destroy cluster
│   └── port-forward.sh               # Start all port-forwards
```

```
└── docs/
    ├── architecture.md              # Detailed architecture notes
    ├── helm-learning-notes.md       # Notes on what Helm is doing under the hoo
    └── sumo-dual-write.md           # How the dual-write pattern works
```

kind Cluster Configuration

File: `kind/cluster-config.yaml`

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
name: watchtower
nodes:
  - role: control-plane
    extraPortMappings:
      # Grafana UI
      - containerPort: 30000
        hostPort: 3000
        protocol: TCP
      # OTLP gRPC (for instrumented apps to send telemetry)
      - containerPort: 30317
        hostPort: 4317
        protocol: TCP
      # OTLP HTTP
      - containerPort: 30318
        hostPort: 4318
        protocol: TCP
```

Single-node cluster is sufficient for a local POC. The `extraPortMappings` expose services via NodePort so instrumented apps running outside the cluster (on the Mac) can send telemetry to Alloy and access Grafana.

Create the cluster:

```
kind create cluster --config kind/cluster-config.yaml
```

Verify:

```
kubectl cluster-info --context kind-watchtower
kubectl get nodes
```

Helm Deployments

All LGTM components use official Grafana Helm charts. Install the Grafana Helm repo first:

```
helm repo add grafana https://grafana.github.io/helm-charts
helm repo update
```

Namespace

Deploy everything into a dedicated namespace:

```
kubectl create namespace watchtower
```

1. Tempo (Traces)

Chart: `grafana/tempo`

File: `helm/values/tempo.yaml`

```
tempo:
  storage:
    trace:
      backend: local
      local:
        path: /var/tempo/traces
  receivers:
    otlp:
      protocols:
        grpc:
          endpoint: "0.0.0.0:4317"
  resources:
    requests:
      cpu: 100m
      memory: 256Mi
    limits:
      cpu: 500m
      memory: 512Mi

persistence:
  enabled: true
  size: 5Gi


helm install tempo grafana/tempo \
  -n watchtower \
  -f helm/values/tempo.yaml
```

What Helm does here (learning note): This creates a StatefulSet with a PersistentVolumeClaim for trace storage, a Service for internal discovery (`tempo.watchtower.svc.cluster.local:3200`), a ConfigMap with the Tempo YAML config rendered from these values, and a ServiceAccount. Run `helm template tempo grafana/tempo -f helm/values/tempo.yaml` to see the raw manifests.

2. Loki (Logs)

Chart: `grafana/loki`

File: `helm/values/loki.yaml`

```
deploymentMode: SingleBinary

loki:
  auth_enabled: false
  commonConfig:
    replication_factor: 1
  storage:
    type: filesystem
  schemaConfig:
    configs:
      - from: "2024-01-01"
        store: tsdb
        object_store: filesystem
        schema: v13
        index:
          prefix: index_
          period: 24h

singleBinary:
  replicas: 1
  resources:
    requests:
      cpu: 100m
      memory: 256Mi
    limits:
      cpu: 500m
      memory: 512Mi
  persistence:
    enabled: true
    size: 5Gi

# Disable components not needed in single-binary mode
backend:
  replicas: 0
```

```
read:
  replicas: 0
write:
  replicas: 0

gateway:
  enabled: false

# Disable self-monitoring to reduce resource usage
monitoring:
  selfMonitoring:
    enabled: false
  lokiCanary:
    enabled: false


helm install loki grafana/loki \
  -n watchtower \
  -f helm/values/loki.yaml
```

What Helm does here: Loki's chart is complex — it supports microservices mode (read/write/backend split), simple scalable mode, and single-binary. We're using single-binary to keep resource usage low. The chart still generates a surprising amount of YAML. Use `helm template` to see it.

3. Mimir (Metrics)

Chart: `grafana/mimir-distributed`

File: `helm/values/mimir.yaml`

```
mimir:
  structuredConfig:
    common:
      storage:
        backend: filesystem
        filesystem:
          dir: /data/mimir
    blocks_storage:
      backend: filesystem
      filesystem:
        dir: /data/mimir/blocks
    ingester:
      ring:
        replication_factor: 1
```

```
# Run everything in a single zone for local dev
ingester:
  replicas: 1
  resources:
    requests:
      cpu: 100m
      memory: 256Mi
    limits:
      cpu: 500m
      memory: 512Mi
  persistentVolume:
    enabled: true
    size: 5Gi
  zoneAwareReplication:
    enabled: false

distributor:
  replicas: 1
  resources:
    requests:
      cpu: 50m
      memory: 128Mi
    limits:
      cpu: 250m
      memory: 256Mi

querier:
  replicas: 1
  resources:
    requests:
      cpu: 50m
      memory: 128Mi
    limits:
      cpu: 250m
      memory: 256Mi

query_frontend:
  replicas: 1
  resources:
    requests:
      cpu: 50m
      memory: 128Mi
    limits:
      cpu: 250m
      memory: 256Mi

store_gateway:
```

```
  replicas: 1
  resources:
    requests:
      cpu: 50m
      memory: 128Mi
    limits:
      cpu: 250m
      memory: 256Mi
  persistentVolume:
    enabled: true
    size: 5Gi
  zoneAwareReplication:
    enabled: false

compactor:
  replicas: 1
  resources:
    requests:
      cpu: 50m
      memory: 128Mi
    limits:
      cpu: 250m
      memory: 256Mi
  persistentVolume:
    enabled: true
    size: 5Gi

# Disable components not needed locally
alertmanager:
  enabled: false
ruler:
  enabled: false
overrides_exporter:
  enabled: false
nginx:
  enabled: false

# Disable minio — we use local filesystem
minio:
  enabled: false


helm install mimir grafana/mimir-distributed \
  -n watchtower \
  -f helm/values/mimir.yaml
```

What Helm does here: mimir-distributed is the most complex chart. Even with everything

set to 1 replica, you still get distributor, ingester, querier, query-frontend, store-gateway, and compactor as separate Deployments/StatefulSets. This is actually good for learning — it shows how a distributed system decomposes, even when running on one node.

4. Grafana (Dashboards)

Chart: `grafana/grafana`

File: `helm/values/grafana.yaml`

```
adminUser: admin
adminPassword: watchtower

service:
  type: NodePort
  nodePort: 30000

resources:
  requests:
    cpu: 50m
    memory: 128Mi
  limits:
    cpu: 250m
    memory: 256Mi

datasources:
  datasources.yaml:
    apiVersion: 1
    datasources:
      - name: Tempo
        type: tempo
        access: proxy
        url: http://tempo.watchtower.svc.cluster.local:3200
        isDefault: false
        jsonData:
          tracesToLogsV2:
            datasourceUid: loki
            filterByTraceID: true
          tracesToMetrics:
            datasourceUid: mimir
          serviceMap:
            datasourceUid: mimir
      - name: Loki
        type: loki
        access: proxy
        url: http://loki.watchtower.svc.cluster.local:3100
```

```
        isDefault: false
        jsonData:
          derivedFields:
            - datasourceUid: tempo
              matcherRegex: '"trace_id":"(\w+)"'
              name: TraceID
              url: '$${__value.raw}'
      - name: Mimir
        type: prometheus
        access: proxy
        url: http://mimir-query-frontend.watchtower.svc.cluster.local:8080/promet
        isDefault: true
        jsonData:
          exemplarTraceIdDestinations:
            - datasourceUid: tempo
              name: trace_id

persistence:
  enabled: true
  size: 1Gi


helm install grafana grafana/grafana \
  -n watchtower \
  -f helm/values/grafana.yaml
```

What Helm does here: The datasources block is provisioned automatically on startup via Grafana's provisioning API. The cross-references between datasources (traces → logs, traces → metrics, logs → traces) are the key value — this is what makes Grafana's "correlated observability" work. Click a trace, see the logs. Click a log line with a trace_id, jump to the trace.

5. Grafana Alloy (Collector + Router)

Chart: `grafana/alloy`

File: `helm/values/alloy.yaml`

```
alloy:
  configMap:
    create: false
    name: alloy-config
    key: config.alloy

controller:
  type: deployment
```

```
  replicas: 1

service:
  type: NodePort

resources:
  requests:
    cpu: 100m
    memory: 128Mi
  limits:
    cpu: 500m
    memory: 256Mi
```

The Alloy config is managed separately (see next section) because it changes more frequently than the Helm values. We create it as a ConfigMap before deploying the chart.

Before deploying, set your Sumo Logic OTLP prototype endpoint. Create a Kubernetes Secret for it:

```
kubectl create secret generic sumo-credentials \
  -n watchtower \
  --from-literal=endpoint=https://your-sumo-otlp-endpoint.sumologic.com/...
```

Then create the ConfigMap and deploy:

```
# Default is dual-write (LGTM + Sumo Logic):
kubectl create configmap alloy-config \
  -n watchtower \
  --from-file=config.alloy=alloy/config.alloy

# Then deploy:
helm install alloy grafana/alloy \
  -n watchtower \
  -f helm/values/alloy.yaml
```

If the Sumo endpoint isn’t ready yet, use the local-only fallback:

```
kubectl create configmap alloy-config \
  -n watchtower \
  --from-file=config.alloy=alloy/config-local-only.alloy

helm install alloy grafana/alloy \
  -n watchtower \
  -f helm/values/alloy.yaml
```

To switch between modes at any time, use `make disable-local-only` or `make enable-local-only`.

Alloy Configuration

Default: Dual-Write (LGTM + Sumo Logic)

File: `alloy/config.alloy`

The default configuration dual-writes all telemetry to both the local LGTM stack and a Sumo Logic OTLP endpoint. This validates the fan-out pattern from day one using synthetic data against a dedicated Sumo prototype collector.

```
// ===== OTLP Receiver =====
// Receives telemetry from instrumented applications via gRPC and HTTP
// Fan-out: same data enters two independent pipelines

otelcol.receiver.otlp "default" {
  grpc {
    endpoint = "0.0.0.0:4317"
  }
  http {
    endpoint = "0.0.0.0:4318"
  }

  output {
    traces  = [otelcol.processor.batch.local.input, otelcol.processor.batch.sumo.
    metrics = [otelcol.processor.batch.local.input, otelcol.processor.batch.sumo.
    logs    = [otelcol.processor.batch.local.input, otelcol.processor.batch.sumo.
  }
}

// ===== Local LGTM Pipeline =====

otelcol.processor.batch "local" {
  timeout = "5s"
  send_batch_size = 1000

  output {
    traces  = [otelcol.exporter.otlphttp.tempo.input]
    metrics = [otelcol.exporter.prometheusremotewrite.mimir.input]
    logs    = [otelcol.exporter.loki.default.input]
  }
}
```

```
otelcol.exporter.otlphttp "tempo" {
  client {
    endpoint = "http://tempo.watchtower.svc.cluster.local:4318"
  }
}

otelcol.exporter.prometheusremotewrite "mimir" {
  endpoint {
    url = "http://mimir-distributor.watchtower.svc.cluster.local:8080/api/v1/push
  }
}

otelcol.exporter.loki "default" {
  forward_to = [loki.write.default.receiver]
}

loki.write "default" {
  endpoint {
    url = "http://loki.watchtower.svc.cluster.local:3100/loki/api/v1/push"
  }
}

// ===== Sumo Logic Pipeline =====
// Sends the same telemetry to a dedicated Sumo Logic OTLP prototype collector
// Set SUMO_OTLP_ENDPOINT via environment variable or Kubernetes Secret

otelcol.processor.batch "sumo" {
  timeout = "10s"
  send_batch_size = 2000

  output {
    traces  = [otelcol.exporter.otlphttp.sumo.input]
    metrics = [otelcol.exporter.otlphttp.sumo.input]
    logs    = [otelcol.exporter.otlphttp.sumo.input]
  }
}

otelcol.exporter.otlphttp "sumo" {
  client {
    endpoint = env("SUMO_OTLP_ENDPOINT")
    // If Sumo requires headers for authentication:
    // headers = {
    //   "X-Sumo-Source" = env("SUMO_SOURCE_HEADER"),
    // }
  }
}
```

How the dual-write works: The fan-out happens at the receiver output level — the same telemetry enters two independent batch processors with separate exporters. If Sumo Logic is slow or unavailable, the local pipeline is unaffected (and vice versa). Alloy handles backpressure per-exporter independently.

Fallback: Local LGTM Only

File: `alloy/config-local-only.alloy`

Use this if the Sumo prototype endpoint is unavailable or you want to reduce noise during initial setup.

```
// ===== OTLP Receiver =====

otelcol.receiver.otlp "default" {
  grpc {
    endpoint = "0.0.0.0:4317"
  }
  http {
    endpoint = "0.0.0.0:4318"
  }

  output {
    traces  = [otelcol.processor.batch.default.input]
    metrics = [otelcol.processor.batch.default.input]
    logs    = [otelcol.processor.batch.default.input]
  }
}

// ===== Batch Processor =====

otelcol.processor.batch "default" {
  timeout = "5s"
  send_batch_size = 1000

  output {
    traces  = [otelcol.exporter.otlphttp.tempo.input]
    metrics = [otelcol.exporter.prometheusremotewrite.mimir.input]
    logs    = [otelcol.exporter.loki.default.input]
  }
}

// ===== Trace Export → Tempo =====

otelcol.exporter.otlphttp "tempo" {
  client {
```

```
    endpoint = "http://tempo.watchtower.svc.cluster.local:4318"
  }
}

// ===== Metrics Export → Mimir =====

otelcol.exporter.prometheusremotewrite "mimir" {
  endpoint {
    url = "http://mimir-distributor.watchtower.svc.cluster.local:8080/api/v1/push
  }
}

// ===== Logs Export → Loki =====

otelcol.exporter.loki "default" {
  forward_to = [loki.write.default.receiver]
}

loki.write "default" {
  endpoint {
    url = "http://loki.watchtower.svc.cluster.local:3100/loki/api/v1/push"
  }
}
```

Synthetic Test Data Generator

File: `test-data/generate.py`

A Python script that generates realistic OTLP telemetry (traces, metrics, logs) and sends it to Alloy. This validates the full pipeline without needing a real instrumented application.

```
test-data/
├── generate.py           # Main generator script
├── requirements.txt      # opentelemetry-api, opentelemetry-sdk,
│                         # opentelemetry-exporter-otlp-proto-grpc
└── README.md
```

The generator should simulate a simple multi-service application:

- **Service: api-gateway** — receives HTTP requests, forwards to backend
- **Service: order-service** — processes orders, writes to database
- **Service: payment-service** — handles payment processing

Each simulated request produces:

- A distributed trace with spans across all three services
- Metrics: request count, latency histogram, error rate
- Structured log lines correlated to the trace via `trace_id`

The generator should run in a loop, producing ~10 requests/second with randomized latency and a ~5% error rate, so dashboards have interesting data to visualize.

Requirements:

```
opentelemetry-api>=1.20.0
opentelemetry-sdk>=1.20.0
opentelemetry-exporter-otlp-proto-grpc>=1.20.0
```

Usage:

```
cd test-data
pip install -r requirements.txt
python generate.py --endpoint localhost:4317 --rate 10
```

The endpoint defaults to `localhost:4317` which hits Alloy via the NodePort mapping.

Makefile

File: `Makefile`

```
.PHONY: setup teardown deploy status port-forward logs test-data render

# === Cluster lifecycle ===

setup:
	kind create cluster --config kind/cluster-config.yaml
	kubectl create namespace watchtower
	@echo "Cluster 'watchtower' created. Run 'make deploy' to install the LGT

teardown:
	kind delete cluster --name watchtower

# === Deploy LGTM stack ===

deploy: deploy-tempo deploy-loki deploy-mimir deploy-grafana deploy-alloy
	@echo "All components deployed. Run 'make status' to check pod health."
	@echo "Run 'make port-forward' to access Grafana at http://localhost:3000
```

```
deploy-tempo:
	helm upgrade --install tempo grafana/tempo \
		-n watchtower \
		-f helm/values/tempo.yaml

deploy-loki:
	helm upgrade --install loki grafana/loki \
		-n watchtower \
		-f helm/values/loki.yaml

deploy-mimir:
	helm upgrade --install mimir grafana/mimir-distributed \
		-n watchtower \
		-f helm/values/mimir.yaml

deploy-grafana:
	helm upgrade --install grafana grafana/grafana \
		-n watchtower \
		-f helm/values/grafana.yaml

deploy-alloy:
	kubectl create configmap alloy-config \
		-n watchtower \
		--from-file=config.alloy=alloy/config.alloy \
		--dry-run=client -o yaml | kubectl apply -f -
	helm upgrade --install alloy grafana/alloy \
		-n watchtower \
		-f helm/values/alloy.yaml

# === Switch Alloy to local-only mode (if Sumo endpoint is unavailable) ===

enable-local-only:
	kubectl create configmap alloy-config \
		-n watchtower \
		--from-file=config.alloy=alloy/config-local-only.alloy \
		--dry-run=client -o yaml | kubectl apply -f -
	kubectl rollout restart deployment alloy -n watchtower
	@echo "Alloy reverted to local LGTM only."

disable-local-only:
	kubectl create configmap alloy-config \
		-n watchtower \
		--from-file=config.alloy=alloy/config.alloy \
		--dry-run=client -o yaml | kubectl apply -f -
	kubectl rollout restart deployment alloy -n watchtower
	@echo "Alloy restored to dual-write (LGTM + Sumo Logic)."
```

```
# === Operations ===

status:
	kubectl get pods -n watchtower -o wide
	@echo ""
	kubectl get svc -n watchtower

port-forward:
	@echo "Starting port-forwards (Ctrl+C to stop)..."
	@echo "Grafana:  http://localhost:3000  (admin / watchtower)"
	@echo "OTLP:    localhost:4317 (gRPC), localhost:4318 (HTTP)"
	@./scripts/port-forward.sh

logs:
	kubectl logs -n watchtower -l app.kubernetes.io/name=alloy -f --tail=50

# === Test data ===

test-data:
	cd test-data && pip install -r requirements.txt --break-system-packages &
		python generate.py --endpoint localhost:4317 --rate 10

# === Learning: render Helm templates to see raw manifests ===

render:
	@mkdir -p helm/rendered
	helm template tempo grafana/tempo -f helm/values/tempo.yaml > helm/render
	helm template loki grafana/loki -f helm/values/loki.yaml > helm/rendered/
	helm template mimir grafana/mimir-distributed -f helm/values/mimir.yaml >
	helm template grafana grafana/grafana -f helm/values/grafana.yaml > helm/
	helm template alloy grafana/alloy -f helm/values/alloy.yaml > helm/render
	@echo "Rendered manifests written to helm/rendered/"
	@echo "Open these files to see what Helm generates under the hood."
```

Phased Build Plan

Phase 1: Cluster + Grafana (Day 1)

Goal: Get a kind cluster running with Grafana accessible in the browser.

1. Install prerequisites: Docker Desktop, kind, kubectl, helm
2. Create the cluster with `make setup`
3. Deploy only Grafana: `make deploy-grafana`

4. Port-forward and verify Grafana loads at `http://localhost:3000`
5. Explore the Grafana UI, poke around settings

K8s concepts learned: Clusters, nodes, namespaces, pods, services, NodePort, port-forwarding, `kubectl get/describe/logs` .

Phase 2: Tempo + Traces (Day 2)

Goal: Send a trace and see it in Grafana.

1. Deploy Tempo: `make deploy-tempo`
2. Verify Tempo pod is healthy
3. Write a minimal Python script that sends one trace to `localhost:4317` (or use the test data generator in trace-only mode)
4. Open Grafana → Explore → Tempo → search for the trace
5. See the waterfall view

K8s concepts learned: StatefulSets, PersistentVolumeClaims, ConfigMaps, service discovery (how Grafana finds Tempo via `tempo.watchtower.svc.cluster.local`).

Phase 3: Loki + Logs (Day 3)

Goal: Send structured logs correlated to traces.

1. Deploy Loki: `make deploy-loki`
2. Extend the test script to emit logs with `trace_id` in the JSON
3. Open Grafana → Explore → Loki → query for logs
4. Click a trace_id link in a log line → jump to Tempo
5. Verify the Tempo → Loki correlation works in the other direction

K8s concepts learned: Different deployment modes (single-binary vs microservices), why `auth_enabled: false` matters locally, log storage schemas.

Phase 4: Mimir + Metrics (Day 4)

Goal: Send Prometheus-format metrics and build a dashboard.

1. Deploy Mimir: `make deploy-mimir`
2. Extend the test script to emit metrics (request count, latency histogram)

3. Open Grafana → Explore → Mimir (Prometheus datasource) → query metrics
4. Build a simple dashboard: request rate, p50/p95/p99 latency, error rate
5. Verify exemplars link metrics to traces

K8s concepts learned: Distributed system components (distributor, ingester, querier, compactor), resource limits, how Helm values map to component configuration.

Phase 5: Alloy + Full Pipeline (Day 5)

Goal: Replace direct OTLP exports with Alloy as the central collector.

1. Deploy Alloy: `make deploy-alloy`
2. Point the test data generator at Alloy instead of individual backends
3. Verify traces, logs, and metrics all flow through Alloy → backends → Grafana
4. Run `make render` and study the raw manifests
5. Experiment: kill a pod (`kubectl delete pod ...`) and watch it recover

K8s concepts learned: DaemonSets vs Deployments, ConfigMap hot-reloading, health checks, pod restart policies, resource requests vs limits.

Phase 6: Dual-Write to Sumo Logic (Week 2)

Goal: Validate the fan-out pattern with real Sumo Logic endpoints.

1. Configure Sumo Logic OTLP endpoint credentials
2. Run `make enable-dual-write`
3. Verify telemetry appears in both Grafana and Sumo Logic
4. Compare query capabilities between the two platforms
5. Document findings for team evaluation

This phase is optional for the POC. It proves the migration pattern works but requires Sumo Logic OTLP endpoint access.

Resource Budget

Estimated resource usage on the MacBook Pro with all components running:

Component	CPU Request	Memory Request	CPU Limit	Memory Limit

Tempo	100m	256Mi	500m	512Mi
Loki	100m	256Mi	500m	512Mi
Mimir (6 pods)	350m	768Mi	1500m	1536Mi
Grafana	50m	128Mi	250m	256Mi
Alloy	100m	128Mi	500m	256Mi
Total	**700m**	**~1.5Gi**	**3250m**	**~3Gi**

This fits comfortably on a MacBook Pro with 8GB allocated to Docker Desktop. The limits allow bursting during query-heavy periods without starving the host OS.

Out of Scope (Future Work)

- **AWS EKS deployment** — CDK infrastructure, IRSA, S3-backed storage for Tempo/Loki/Mimir
- **Multi-region** — deploying to multiple AWS regions
- **Datadog migration** — dual-write config for Datadog in addition to Sumo Logic
- **Alerting** — Grafana alerting rules, Alertmanager integration
- **Authentication** — Grafana OIDC/SSO, Alloy mTLS
- **Argus integration** — connecting Claude Code / MCP to query the LGTM APIs
- **Production hardening** — HA configurations, backup/restore, retention policies
- **Real application telemetry** — replacing synthetic data with actual work application instrumentation

These are all natural next steps once the local POC validates the architecture and the team is comfortable with the K8s workflow. The Alloy dual-write pattern means adding new export destinations (Datadog, Grafana Cloud, etc.) is a config change, not an architecture change.

Notes for Claude Code

This spec is designed to be fed into Claude Code for implementation. Key points:

- Start with `make setup` and work through the phases sequentially

- Each phase should be validated (pods healthy, data visible in Grafana) before moving to the next
- The Helm values files are intentionally minimal — only override what's needed for local dev
- The Alloy config files are the most likely to need iteration as you discover routing quirks
- Use `kubectl describe pod <name> -n watchtower` when pods aren't starting — the Events section usually tells you why
- Use `kubectl logs <pod> -n watchtower` to debug application-level issues
- If a Helm install fails, use `helm uninstall <name> -n watchtower` and try again
- The `make render` target is for learning — read the rendered YAML to understand what Helm abstracts away