SRE Demo Cheat Sheet

Quick Reference for Live Demonstration

Pre-Demo Setup (10 minutes before)

```
# 1. Start all services
docker compose -f docker-compose.yml -f docker/docker-compose.caddy.yml up -d
sleep 30

# 2. Start load generator
./generate-load.sh 3600 30 > /tmp/load-gen.log 2>&1 &
echo $! > /tmp/load-gen.pid

# 3. Verify services
docker compose ps
curl http://localhost:8080/health
curl http://localhost:3000/api/health

# 4. Open Grafana
xdg-open http://localhost:3000  # Login: admin/admin123

# 5. Generate baseline flamegraph
make profile-cpu
```

Demo Flow Timeline

Time	Section	Command
0-5 min	Introduction	(Talking only)
5-10 min	Environment Tour	Show Grafana + `docker compose ps`
10-20 min	Flamegraph Demo	`make profile-cpu`
20-30 min	Chaos: Network	`make chaos-network-partition`
30-40 min	Chaos: CPU	`make chaos-cpu-throttle`
40-45 min	Wrap-up	(Talking + Q&A)

Phase 1: Introduction (5 min)

No commands - just talking

Key points: - Problem: Issues found in production, hard to debug - Solution: Chaos + Flamegraphs + eBPF - What you'll see: Real failures, real profiling, kernel tracing

---

Phase 2: Environment Tour (5 min)

Show Running Services

```
docker compose ps
```

Show Resource Usage

```
docker stats --no-stream
```

Show Current Metrics

```
# Order rate
curl -s --data-urlencode 'query=rate(orders_total[1m])' \
  http://localhost:9090/api/v1/query | jq '.data.result[0].value[1]'

# Latency
curl -s http://localhost:8080/health | jq '.latency'
```

Navigate Grafana

- Open: http://localhost:3000
- Show: System Overview dashboard
- Point out: Order rate, latency, service health

Phase 3: Flamegraph Profiling (10 min)

Generate CPU Flamegraph

```
make profile-cpu
```

Wait 30 seconds while it profiles

When flamegraph opens: 1. Explain: Width = CPU time, Height = call stack 2. Find: Widest boxes (hot paths) 3. Hover: Show percentages 4. Point out: Database ~40%, JSON ~12%

Generate All Profiles

```
make profile-all
```

Explain types: - CPU: Where time is spent - Heap: Memory allocations - Goroutine: Concurrency - Mutex: Lock contention

---

Phase 4: Network Partition Chaos (10 min)

Setup

Position screens: - **Left**: Grafana dashboard - **Right**: Terminal

Execute

```
make chaos-network-partition
```

Narration Points

When it starts: > “Setting up eBPF to trace TCP connections…”

When baseline shows: > “Capturing normal metrics: 30 orders/min, 125ms latency”

When failure injected (at ~30s): > “NOW - watch Grafana - connections will start failing”

When eBPF traces show (at ~45s-90s): > “See TCP retransmissions? That’s the kernel trying to reconnect. eBPF shows us kernel behavior without touching our code.”

Point to Grafana: - Error rate spikes to 100% - Latency shows timeouts - Database connections drop to 0 - Health indicator turns red

When recovery starts (at ~90s): > “Removing partition… watch the recovery”

Point to Grafana: - Errors drop immediately - Connections re-establish - Latency returns to normal

When summary shows: > “5.8 seconds to full recovery, zero orders lost”

Key Takeaways

- eBPF traced kernel networking
- Auto-recovery validated
- Retry logic working correctly
- Recovery time measured

Phase 5: CPU Throttling Chaos (10 min)

Execute

```
make chaos-cpu-throttle
```

Narration Points

When baseline shows: > "Normal CPU usage: 25%, latency 120ms"

When CPU limited (at ~10s): > "Limiting to 10% CPU - watch latency climb in Grafana"

During degradation (10s-60s): > "Latency increasing… 250ms… 500ms… 1000ms… but no failures"

Point to eBPF traces (at ~30s): > "Scheduler delays visible: 90% of time spent waiting for CPU quota"

Point to Grafana: - Latency climbs steadily - Queue depth increases - CPU pegged at 100% (of limit) - No errors - graceful degradation

When recovery starts (at ~60s): > "Removing limit… watch queue drain"

Point to Grafana: - Latency drops rapidly - Queue processes - Back to baseline in 20 seconds

When summary shows: > "10x latency increase but 0% errors - system degrades gracefully, not catastrophically"

Key Takeaways

- eBPF traced scheduler delays
- Graceful degradation confirmed
- Capacity measured: 3x current load
- Recovery time known

Phase 6: Wrap-Up (5 min)

Summary

What we showed: 1. Flamegraphs: Visual performance analysis (40% DB, 12% JSON) 2. Network chaos: eBPF traced TCP, 5.8s recovery 3. CPU chaos: Graceful degradation, capacity measured

Business value: - Find issues before customers - Optimize costs (40% DB → optimization opportunity) - Debug production safely (eBPF <1% overhead)

ROI: 2 weeks investment, 6 month payback

Next Steps

1. Run remaining experiments (OOM, disk I/O, cascade)
2. Review flamegraphs for optimization
3. Team training on bpftrace
4. Integrate into CI/CD

Emergency Commands

Stop Everything

```
# Kill load generator
kill $(cat /tmp/load-gen.pid 2>/dev/null)

# Stop demo
pkill -f chaos-experiments

# Restart services
docker compose restart
```

Cleanup Chaos Mess

```
# Clear iptables rules
sudo iptables -F DOCKER

# Reset CPU limits
docker update --cpus=2.0 simulated-exchange-trading-api

# Restart all
docker compose restart
```

Quick Recovery

```
# Nuclear option
docker compose down
docker compose -f docker-compose.yml -f docker/docker-compose.caddy.yml up -d
sleep 60
./generate-load.sh 3600 30 &
```

Useful Queries During Demo

Prometheus Queries

```
# Order rate
curl -s --data-urlencode 'query=rate(orders_total[1m])' \
  http://localhost:9090/api/v1/query | jq -r '.data.result[0].value[1]'

# Error rate
curl -s --data-urlencode 'query=rate(orders_total{status="FAILED"}[1m])' \
  http://localhost:9090/api/v1/query | jq -r '.data.result[0].value[1] // "0"'
```

```
# Database connections
curl -s --data-urlencode 'query=database_connections_active' \
  http://localhost:9090/api/v1/query | jq -r '.data.result[0].value[1]'
```

Health Checks

```
# Trading API
curl -s http://localhost:8080/health | jq .

# All services
for port in 8080 8081 8082; do
  echo "Port $port:"
  curl -s http://localhost:$port/health | jq -r '.status // "N/A"'
done
```

Talking Points

For Management

Cost savings: > “40% of CPU in database - optimization could reduce cloud costs by 30-40%”

Risk reduction: > “Finding these issues in testing, not production. Netflix saved millions with chaos engineering.”

Competitive advantage: > “Industry leaders (Google, Amazon, Netflix) use these exact techniques”

For Engineers

Learning value: > “eBPF shows what’s really happening at kernel level - best debugging education you can get”

Safety: > “All experiments auto-cleanup. Safe to run in staging.”

Actionable: > “Not just pretty graphs - specific optimizations identified (database, JSON serialization)”

Common Questions & Answers

Q: Is this safe for production? > A: Start in staging. Some experiments (CPU throttle, flamegraphs) are safe for prod. Others (network partition, OOM) are staging-only.

Q: What’s the overhead? > A: Flamegraphs: <5% CPU. eBPF: <1% CPU. Both safe for production.

Q: How long to implement? > A: 2 weeks for team training and integration. ROI in 6 months via incident reduction and cost optimization.

Q: What if we don't use Go? > A: Flamegraphs work with any language. eBPF is language-agnostic. Tools differ but concepts are identical.

Q: Can we do this in our environment? > A: Yes. Everything shown is open source and works on any Linux system with kernel 4.9+.

Post-Demo Cleanup

```
# 1. Stop load generator
kill $(cat /tmp/load-gen.pid)
rm /tmp/load-gen.pid

# 2. Archive artifacts
mkdir -p ~/demo-$(date +%Y%m%d)
cp -r flamegraphs ~/demo-$(date +%Y%m%d)/
cp -r chaos-results ~/demo-$(date +%Y%m%d)/
cp /tmp/chaos-exp-*.log ~/demo-$(date +%Y%m%d)/ 2>/dev/null

# 3. Optional: Stop services
docker compose down

# 4. Create artifact package
cd ~/demo-$(date +%Y%m%d)
tar -czf ../demo-artifacts-$(date +%Y%m%d).tar.gz .
```

URLs to Have Open

- Grafana: http://localhost:3000 (admin/admin123)
- Prometheus: http://localhost:9090
- Trading API: http://localhost:8080
- This cheatsheet: `docs/SRE_DEMO_CHEATSHEET.md`
- Full guide: `docs/SRE_DEMO_GUIDE.md`

Screen Layout Recommendation

```
  MONITOR 1 (Audience View)

      Grafana                    Flamegraph
```

```
    Dashboard            (Browser)


    Terminal (chaos experiments)




MONITOR 2 (Your View)

   Cheatsheet           Notes
   (This file)
```

Good luck with the demo!