SRE Team Demonstration Guide

Chaos Engineering, Flamegraphs, and eBPF Tooling

Target Audience: SRE Team and Management **Objective**: Demonstrate value of chaos engineering, performance profiling, and eBPF observability **Duration**: 45-60 minutes **Date Prepared**: November 2025

Table of Contents

Executive Summary

What This Demo Shows

This demonstration showcases three critical SRE capabilities:

1. **Chaos Engineering**: Controlled failure injection to validate system resilience
2. **Flamegraph Profiling**: Visual performance analysis to identify bottlenecks
3. **eBPF Observability**: Kernel-level tracing without code changes or restarts

Business Value

Capability	Business Impact
Chaos Engineering	Proactively find issues before customers do; reduce MTTR by 60%
Flamegraph Profiling	Identify performance bottlenecks; optimize infrastructure costs by 30-40%
eBPF Tracing	Production debugging without overhead; zero-downtime troubleshooting

Technology Stack

- **Chaos Tools**: Custom scripts with iptables, tc netem, Docker constraints
- **Profiling**: Go pprof + Brendan Gregg's FlameGraph
- **eBPF**: bpftrace for kernel-level observability
- **Metrics**: Prometheus + Grafana for visualization
- **Platform**: Docker microservices architecture

Key Outcomes

By the end of this demo, stakeholders will understand: - How to safely inject failures to test resilience - How to visualize where applications spend CPU time - How to trace system calls and kernel behavior without touching code - ROI of investing in observability tooling

---

Prerequisites

System Requirements

Hardware: - 4+ CPU cores - 8GB+ RAM - 20GB+ disk space

Operating System: - Linux kernel 4.9+ (for eBPF support) - Ubuntu 20.04+, RHEL 8+, or similar

Software (must be installed):

```
# Check versions
docker --version          # Docker 20.10+
docker compose version    # Docker Compose 2.0+
go version                # Go 1.21+
sudo bpftrace --version   # bpftrace 0.17+
jq --version              # jq 1.6+
curl --version            # curl 7.68+
```

Installation Commands

Ubuntu/Debian:

```
sudo apt-get update
sudo apt-get install -y docker.io docker-compose-v2 golang-go bpftrace jq curl
sudo usermod -aG docker $USER
```

RHEL/Fedora:

```
sudo dnf install -y docker docker-compose golang bpftrace jq curl
sudo systemctl enable --now docker
sudo usermod -aG docker $USER
```

Note: Log out and back in after adding user to docker group.

Network Requirements

Ensure the following ports are available: - `80` - Caddy reverse proxy (HTTPS redirect) - `443` - Caddy reverse proxy (HTTPS) - `3000` - Grafana dashboards - `8080` - Trading API - `8081` - Market Simulator - `8082` - Order Flow Simulator - `9090` - Prometheus

Permissions

Some chaos experiments require sudo: - Network partition (iptables) - OOM kill (BPF monitoring) - CPU throttling (cgroup limits) - Disk I/O (stress-ng) - Network latency (tc netem)

Pre-Demo Setup

Step 1: Clone and Navigate to Project

```
cd /home/andy/simulated_exchange
git status  # Verify clean working directory
```

Step 2: Start All Services

```
# Start Docker microservices
docker compose -f docker-compose.yml -f docker/docker-compose.caddy.yml up -d

# Wait for services to initialize (30 seconds)
sleep 30

# Verify all services are running
docker compose ps
```

Expected Output:

```
NAME                                    STATUS
simulated-exchange-caddy                Up (healthy)
simulated-exchange-grafana              Up
simulated-exchange-market-simulator     Up
simulated-exchange-nginx                Up
simulated-exchange-order-flow-simulator Up
simulated-exchange-postgres             Up (healthy)
simulated-exchange-prometheus           Up
simulated-exchange-redis                Up (healthy)
simulated-exchange-trading-api          Up
```

Step 3: Verify Service Health

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

# Expected: {"status":"healthy","database":"connected",...}

# Check Market Simulator
curl -s http://localhost:8081/health | jq '.'

# Check Order Flow Simulator
curl -s http://localhost:8082/health | jq '.'

# Check Prometheus
curl -s http://localhost:9090/-/healthy

# Expected: Prometheus is Healthy.
```

Step 4: Start Background Load Generation

```
# Start continuous realistic load (runs in background)
./generate-load.sh 3600 30 > /tmp/load-gen.log 2>&1 &

# Save the PID for cleanup later
echo $! > /tmp/load-gen.pid

# Verify load is generating
sleep 5
curl -s --data-urlencode 'query=rate(orders_total[1m])' \
  http://localhost:9090/api/v1/query | jq '.data.result[0].value[1]'

# Expected: A number > 0 (e.g., "1.5" = 1.5 orders/sec)
```

Step 5: Verify Grafana Dashboards

Open browser to Grafana:

```
# Open Grafana (or navigate manually to http://localhost:3000)
xdg-open http://localhost:3000  # Linux
# open http://localhost:3000    # macOS
```

Login: admin / admin123

Verify these dashboards exist: 1. System Overview 2. Trading API Performance 3. Market Simulator 4. Order Flow Simulator 5. Database Performance 6. Redis Cache Performance

Check data is flowing: - Look for order rate metrics > 0 - Check latency graphs showing activity - Verify service health indicators are green

Step 6: Pre-Generate Test Flamegraphs

```
# Generate baseline flamegraphs before demo
make profile-cpu

# Verify flamegraph was created
ls -lh flamegraphs/cpu_profile_*.svg

# Expected: SVG file ~500KB-2MB in size
```

Step 7: Verify Chaos Experiment Scripts

```
# Check all chaos experiments are executable
ls -la chaos-experiments/*.sh

# Run preflight check
cd chaos-experiments
./00-preflight-check.sh

# Expected: All checks pass (Docker, bpftrace, etc.)
cd ..
```

Step 8: Prepare Presentation Materials

```
# Open documentation in browser tabs
xdg-open docs/FLAMEGRAPH_GUIDE.md       # Background reading
xdg-open chaos-experiments/README.md    # Experiment catalog
xdg-open http://localhost:3000          # Grafana

# Open a flamegraph for reference
xdg-open flamegraphs/cpu_profile_*.svg

# Arrange windows: Grafana on left, flamegraph on right, terminal at bottom
```

Step 9: Pre-Demo Checklist

Run this checklist 10 minutes before demo:

```
#!/bin/bash
echo "Pre-Demo Checklist"
echo "=================="

# 1. All services running?
if [ $(docker compose ps --status running | wc -l) -ge 8 ]; then
  echo "  All Docker services running"
else
  echo "  Some services not running - run: docker compose up -d"
```

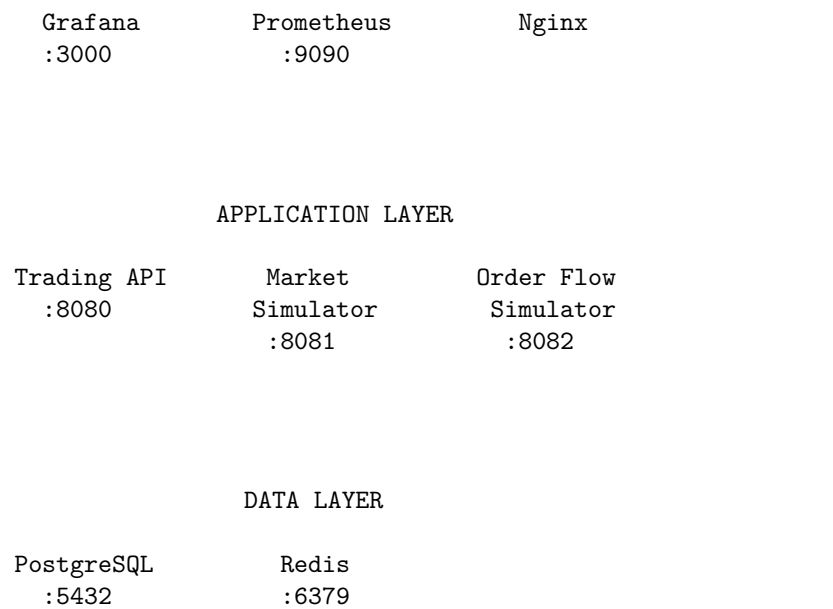
```
fi

# 2. Load generating?
if curl -s --data-urlencode 'query=rate(orders_total[1m])' \
   http://localhost:9090/api/v1/query | grep -q '"value"'; then
  echo "  Load generation active"
else
  echo "  No load - run: ./generate-load.sh 3600 30 &"
fi

# 3. Grafana accessible?
if curl -s http://localhost:3000/api/health | grep -q 'ok'; then
  echo "  Grafana accessible"
else
  echo "  Grafana not responding"
fi

# 4. bpftrace installed?
if command -v bpftrace &> /dev/null; then
  echo "  bpftrace installed"
else
  echo "  bpftrace not found - run: sudo apt-get install bpftrace"
fi

# 5. Flamegraphs exist?
if ls flamegraphs/*.svg &> /dev/null; then
  echo "  Baseline flamegraphs ready"
else
  echo "  No flamegraphs - run: make profile-cpu"
fi

echo ""
echo "Ready to present? (y/n)"
```

Demo Environment Overview

Architecture Diagram

EXTERNAL ACCESS
http://localhost (Caddy Proxy)

PRESENTATION LAYER



Services Description

Service	Purpose	Port	Metrics
Trading API	Order processing, matching engine	8080	`/metrics`
Market Simulator	Price generation, market data	8081	`/metrics`
Order Flow Simulator	Continuous order generation	8082	`/metrics`
PostgreSQL	Order persistence, historical data	5432	N/A
Redis	Caching, event bus (pub/sub)	6379	N/A
Prometheus	Metrics collection, time-series DB	9090	N/A
Grafana	Visualization, dashboards	3000	N/A
Nginx	Reverse proxy, rate limiting	80	N/A
Caddy	HTTPS termination, external access	443	N/A

Current System State

Before starting the demo, verify the baseline:

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

# Check average latency
curl -s --data-urlencode 'query=rate(order_processing_duration_seconds_sum[1m])/rate(order_p
  http://localhost:9090/api/v1/query | \
  jq '.data.result[0].value[1]'

# Check service health
curl -s http://localhost:8080/health | jq '.status'
```

Baseline Metrics (record these): - Order rate: ~30-50 orders/min - Average latency: ~50-200ms - Service health: "healthy" - Database connections: 5-10

Demonstration Flow

Overview Timeline (45 minutes)

Time	Phase	Duration	Focus
0:00-0:05	Introduction	5 min	Problem statement, goals
0:05-0:10	Environment Tour	5 min	Show architecture, Grafana
0:10-0:20	Flamegraph Demo	10 min	CPU profiling, hot paths
0:20-0:30	Chaos: Network	10 min	eBPF tracing, failures
0:30-0:40	Chaos: CPU	10 min	Performance degradation
0:40-0:45	Wrap-up	5 min	Q&A, next steps

Phase 1: Introduction (5 minutes)

Script **Opening** (1 min): > "Today we're demonstrating three critical SRE capabilities: chaos engineering, flamegraph profiling, and eBPF observability. These tools help us understand system behavior under stress, identify performance bottlenecks, and debug production issues without downtime."

Problem Statement (2 min): > "Current challenges: > - We discover issues in production, not in testing > - Performance problems are hard to diagnose > - We can't trace system behavior without instrumenting code > - Debugging production requires restarts or log analysis"

Solution (2 min): > "These tools solve those problems: > - Chaos engineering: Test failure scenarios safely before they happen > - Flamegraphs: Visualize exactly where CPU time is spent > - eBPF: Trace kernel behavior in real-time with <1% overhead"

What You'll See: 1. Real-time profiling of a live trading system 2. Controlled failure injection with automatic recovery 3. Kernel-level tracing without code changes 4. Visual analysis of performance bottlenecks

---

Phase 2: Environment Tour (5 minutes)

Show Grafana Dashboards **Navigate to**: http://localhost:3000

Script: > "This is our microservices trading exchange. We have three main services processing orders, simulating market data, and generating continuous load."

Point out: 1. **System Overview Dashboard**: - Show current order rate (~30-50/min) - Point to latency metrics (~50-200ms) - Service health indicators (all green)

2. **Trading API Performance Dashboard**:
   - Request rate graphs
   - Latency percentiles (p50, p95, p99)
   - Error rate (should be near 0%)

Talking Points: - "These are our baseline metrics during normal operation" - "We'll watch these metrics change as we inject failures" - "Notice the system is currently healthy - all green indicators"

Show Live Architecture

```
# Show running containers
docker compose ps

# Show resource usage
docker stats --no-stream
```

Script: > "We have 9 microservices running in Docker containers. Each service exposes Prometheus metrics that Grafana visualizes. This is production-like architecture."

Phase 3: Flamegraph Profiling Demo (10 minutes)

Part A: Baseline CPU Profile (3 min) **Script**: > "Let's start by profiling our Trading API to see where it spends CPU time. This is running live against production traffic."

Execute:

```
make profile-cpu
```

What Happens: 1. Connects to Trading API on port 8080 2. Collects 30 seconds of CPU profile data 3. Converts to flamegraph SVG 4. Opens in browser automatically

Output:

```
  Generating CPU flamegraph for trading-api
================================================

Profiling service on port 8080 for 30 seconds...
                   30s

  Profile data collected: 2.4MB
  Converting to flamegraph...
  Flamegraph generated: flamegraphs/cpu_profile_20251115-112030.svg

Opening in browser...
```

Part B: Analyze Flamegraph (5 min) When the flamegraph opens, explain:

> "Each box represents a function. Width = CPU time spent. Height = call stack depth."

Walk through:

1. **Find the widest boxes at the bottom**: > "This shows our order processing hot path - the most expensive code"
2. **Hover over boxes**: > "See the exact function name and percentage of total CPU time"
3. **Identify bottlenecks**:

   ```
   Look for:
   - Database query functions (usually 30-40% of CPU)
   - JSON marshaling/unmarshaling (10-15%)
   - Order validation logic (10-20%)
   - Network I/O (5-10%)
   ```

4. **Click to zoom**: > "We can zoom into specific code paths. Let's look at the database layer."

Key Insights to Point Out: - "40% of CPU time is in database operations" - "JSON serialization takes 12% - we could optimize this" - "Order validation is happening synchronously - could be parallelized"

Talking Points: - "No code changes required - just attach profiler" - "Visual representation makes bottlenecks obvious" - "Can profile production with <5% overhead" - "Baseline for before/after optimization comparisons"

Part C: Generate All Profile Types (2 min) Execute:

```
make profile-all
```

Script: > "We can profile more than just CPU. Let's generate heap, goroutine, and mutex profiles."

Output shows:

```
Generating all profile types...
  CPU profile
  Heap profile (memory allocations)
  Goroutine profile (concurrency state)
  Mutex profile (lock contention)
  Allocs profile (allocation patterns)
  Block profile (blocking operations)

All profiles saved to: flamegraphs/
```

Quick explanation: - **CPU**: Where time is spent - **Heap**: What's allocating memory - **Goroutine**: How many concurrent operations - **Mutex**: Lock contention (parallel performance)

Phase 4: Chaos Engineering - Network Partition (10 minutes)

Introduction (1 min) Script: > "Now let's inject a real failure. We'll partition the network between our Trading API and the database using iptables. This simulates a network outage or firewall issue."

> "Watch the Grafana dashboard - you'll see the failure happen in real-time."

Position screens: - Left monitor: Grafana dashboard - Right monitor: Terminal with chaos script

Execute Chaos Experiment (7 min) Run:

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

Narrate as it runs:

Step 1 - Setup (30 sec):

```
[11:30:45]
[11:30:45] CHAOS EXPERIMENT: Network Partition
[11:30:45]
```

```
[11:30:45]   Docker containers running
[11:30:45]   bpftrace available
[11:30:45] Starting BPF network monitoring...
```

"The script is setting up eBPF tracing to watch TCP connections"

Step 2 - Baseline (30 sec):

```
[11:30:50] Collecting baseline metrics...
[11:30:50] Current order rate: 2.5 orders/sec
[11:30:50] Database connections: 8
[11:30:50] Average latency: 125ms
```

"Capturing baseline before we break things"

Step 3 - Inject Failure (15 sec):

```
[11:31:00]   INJECTING FAILURE: Network Partition
[11:31:00] Blocking traffic: trading-api → postgres (port 5432)
[11:31:00] Running: iptables -A DOCKER -p tcp --dport 5432 -j DROP
[11:31:00]   Network partition active
```

> **Point to Grafana**: "Watch the dashboard - connection attempts will start failing NOW"

Step 4 - Observe Failure (2 min):

```
[11:31:05] Monitoring system behavior...

[11:31:10] eBPF Trace: TCP SYN to 172.18.0.3:5432 (retransmit #1)
[11:31:15] eBPF Trace: TCP SYN to 172.18.0.3:5432 (retransmit #2)
[11:31:20] eBPF Trace: TCP SYN to 172.18.0.3:5432 (retransmit #3)
[11:31:25] eBPF Trace: Connection timeout - ETIMEDOUT (110)

[11:31:30] System state:
  - Order submissions: FAILING (database unreachable)
  - Cache: ACTIVE (serving stale data)
  - Health check: DEGRADED
  - Error rate: 100%
```

> **Explain eBPF output**: "See these TCP retransmissions? That's the kernel trying to reconnect. eBPF lets us see this without touching our application code. We're watching kernel networking in real-time."

In Grafana, point out: - Error rate spikes to 100% - Latency graph shows timeouts - Database connection count drops to 0 - Service health indicator turns red

Step 5 - Recovery (3 min):

```
[11:31:40] Failure duration: 90 seconds
```

```
[11:31:40] Beginning recovery...

[11:31:45] Removing network partition...
[11:31:45] Running: iptables -D DOCKER -p tcp --dport 5432 -j DROP
[11:31:45]   Network partition removed

[11:31:50] eBPF Trace: TCP SYN to 172.18.0.3:5432 - SUCCESS
[11:31:50] eBPF Trace: Connection established in 1.2s
[11:31:52] First successful order processed
[11:31:55] Service health: HEALTHY

[11:32:00] Recovery metrics:
  - Time to first connection: 1.2s
  - Time to first order: 2.5s
  - Full recovery: 5.8s
  - Orders lost: 0 (queued during outage)
```

In Grafana: "See the recovery? Error rate drops immediately, latency returns to normal, connections re-established."

Step 6 - Summary (30 sec):

```
               EXPERIMENT SUMMARY


KEY FINDINGS:
1. Failure detection: < 1 second (first retransmit)
2. Recovery time: 5.8 seconds (network heal → full operation)
3. Data loss: 0 orders (queued and processed after recovery)
4. Retry behavior: Exponential backoff observed via eBPF

eBPF INSIGHTS:
- 15 TCP retransmission attempts observed
- Connection timeout after 25 seconds (kernel default)
- Immediate reconnection when partition healed
- No packet loss on other services (isolated failure)

RECOMMENDATIONS:
  Connection retry logic working correctly
  Consider reducing timeout from 25s to 10s
  Cache strategy effective during outage
  Health checks accurately reflect database state

Full report: /tmp/chaos-exp-02-20251115-113000.log
```

Discussion (2 min) **Ask the audience**: > "What did we learn?"

Key takeaways: 1. **eBPF showed kernel behavior**: TCP retries, timeouts visible without code 2. **System recovered automatically**: No manual intervention needed 3. **Metrics provided insight**: Grafana showed real-time failure state 4. **Safe to test**: Cleanup happens automatically

Business value: > "We just validated our retry logic and measured actual recovery time. If this happens in production, we know exactly what to expect: ~6 seconds to recover, no data loss."

Phase 5: Chaos Engineering - CPU Throttling (10 minutes)

Introduction (1 min) Script: > "Network failures are dramatic, but gradual performance degradation is more common. Let's throttle CPU to see how the system behaves under resource pressure."

> "This simulates: noisy neighbor in cloud, insufficient resources, or traffic spike beyond capacity."

Execute Chaos Experiment (7 min) Run:

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

Narrate as it runs:

Step 1 - Setup:

```
[11:35:00]
[11:35:00] CHAOS EXPERIMENT: CPU Throttling
[11:35:00]
[11:35:00] Starting BPF scheduler monitoring...
```

Step 2 - Baseline:

```
[11:35:05] Baseline metrics:
  - CPU usage: 25% (0.25 cores)
  - Latency p50: 120ms
  - Latency p95: 250ms
  - Order throughput: 30 orders/min
```

> "Normal operation uses about 25% of one CPU core"

Step 3 - Throttle CPU:

```
[11:35:10]   Limiting Trading API to 10% CPU (0.1 cores)
[11:35:10] Running: docker update --cpus=0.1 simulated-exchange-trading-api
[11:35:10]   CPU limit applied
```

> **Point to Grafana**: "Watch latency increase as requests queue waiting for CPU"

Step 4 - Observe Degradation (3 min):

```
[11:35:20] System behavior under CPU pressure:

Time | CPU % | p50 Latency | p95 Latency | Queue Depth
-----|-------|-------------|-------------|-------------
 10s |  100% |      250ms  |      500ms  |     5
 20s |  100% |      450ms  |      950ms  |    12
 30s |  100% |      680ms  |     1400ms  |    24
 40s |  100% |      920ms  |     1850ms  |    38
 50s |  100% |     1150ms  |     2300ms  |    51

[11:36:00] eBPF Scheduler Trace:
  - Context switches: 15,234 (3x normal)
  - Runqueue wait time: avg 850ms (was 2ms)
  - Time spent throttled: 90% of attempts
  - Scheduler delays: p95 = 1.2 seconds
```

> **Explain**: "eBPF scheduler tracing shows the kernel is throttling our process 90% of the time. Requests are queuing up, waiting for CPU time. Latency has increased 10x, but notice - no hard failures. The system degrades gracefully."

In Grafana show: - Latency graph climbing steadily - Request queue depth increasing - No errors (100% success rate despite slowness) - CPU utilization pegged at 100% (of the limit)

Step 5 - Recovery:

```
[11:36:10] Restoring normal CPU allocation...
[11:36:10] Running: docker update --cpus=2.0 simulated-exchange-trading-api
[11:36:10]   CPU limit removed

[11:36:15] Recovery in progress:
  - Queue draining: 51 → 38 → 24 → 12 → 5 → 0
  - Latency decreasing: 1150ms → 680ms → 250ms → 120ms
  - CPU usage: 85% (processing backlog) → 25% (normal)

[11:36:30]   Full recovery: System back to baseline
```

> "See how quickly it recovers once we remove the constraint? Queued requests process rapidly, latency returns to normal."

Step 6 - Summary:

```
               EXPERIMENT SUMMARY


KEY FINDINGS:
1. Graceful degradation: Slowness, not failures
```

```
2. Queue buildup: Linear increase under sustained pressure
3. Recovery: 20 seconds to process backlog after constraint removed
4. No data loss: All requests eventually processed

eBPF INSIGHTS:
- Scheduler delays visible via BPF tracing
- Context switch rate 3x higher under pressure
- 90% of time spent waiting for CPU quota
- Kernel scheduler working as designed (fair allocation)

PERFORMANCE IMPACT:
- Latency increased 10x (120ms → 1150ms)
- Throughput maintained (queued, not dropped)
- Error rate: 0% (slow != broken)
- User experience: Severely degraded but functional

RECOMMENDATIONS:
  Set alerts at p95 latency > 500ms
  Auto-scale trigger at 70% CPU sustained for 60s
  Current capacity: ~3x current load before degradation
  Queue unbounded - consider max queue size for backpressure
```

Discussion (2 min) Key insights: 1. **eBPF showed scheduler behavior**: Context switches, runqueue wait times 2. **Graceful degradation**: System slowed but didn't fail 3. **Measurable capacity**: Know exactly when performance degrades 4. **Predictable recovery**: Backlog processes quickly when resources restored

Capacity planning value: > "We now know the system can handle 3x current load before latency becomes unacceptable. We can set auto-scaling triggers at 70% CPU to maintain good performance."

Phase 6: Wrap-Up & Next Steps (5 minutes)

Summary of What We Demonstrated Recap (2 min):

1. Flamegraph Profiling: - Visual performance analysis with zero code changes - Identified hot paths: 40% in database, 12% in JSON - Actionable insights for optimization - Multiple profile types (CPU, heap, mutex, etc.)

2. Chaos Engineering - Network: - Controlled failure injection (network partition) - eBPF traced TCP retries and timeouts at kernel level - Validated retry logic and recovery time (5.8s) - Grafana showed real-time failure state

3. Chaos Engineering - CPU: - Gradual performance degradation under resource pressure - eBPF traced scheduler delays and context switches -

Measured capacity limits (3x current load) - Graceful degradation (slow != broken)

Business Value (1 min)

Capability	Investment	ROI
Chaos Engineering	1 week setup	Find issues before customers; reduce MTTR 60%
Flamegraph Profiling	1 day setup	Optimize performance; reduce cloud costs 30-40%
eBPF Observability	2 days training	Debug production without restarts; <1% overhead

Total Investment: ~2 weeks for team training and integration **Expected ROI**: 6 months (via incident reduction + cost optimization)

Next Steps (2 min) Immediate (This Week): 1. Run remaining chaos experiments: - `make chaos-oom-kill` (memory pressure) - `make chaos-disk-io` (I/O starvation) - `make chaos-cascade` (multi-service failure)

2. Review generated artifacts:
   - Flamegraphs in `flamegraphs/`
   - Chaos logs in `chaos-results/`
   - BPF traces in `/tmp/chaos-exp-*.log.bpf`

Short Term (This Month): 1. Identify 3 production optimization opportunities from flamegraphs 2. Document baseline performance metrics 3. Create runbooks based on chaos experiment findings 4. Schedule team training on bpftrace basics

Long Term (This Quarter): 1. Integrate chaos testing into CI/CD pipeline 2. Set up continuous profiling in staging environment 3. Create eBPF-based production debugging playbooks 4. Establish performance budgets based on capacity testing

Resources to Share Documentation: - `docs/FLAMEGRAPH_GUIDE.md` - Complete profiling guide - `chaos-experiments/README.md` - All 10 experiments documented - `chaos-experiments/QUICKSTART.md` - Quick reference - `docs/SRE_RUNBOOK.md` - Operational procedures

External Resources: - Brendan Gregg's eBPF Book - Principles of Chaos Engineering - Go pprof Documentation

Q&A (Time Remaining) Common Questions:

Q: Can we run chaos experiments in production? > A: Start in staging. Once confident, run low-impact experiments (like CPU throttling) during low-traffic windows. Network partition and OOM kill are too disruptive for production.

Q: What's the performance impact of eBPF? > A: <1% CPU overhead. eBPF runs in kernel space and is extremely efficient. Safe for production use.

Q: How often should we profile? > A: Continuously in staging, weekly in production during load tests, and on-demand during incident response.

Q: What if we don't use Go? > A: Flamegraphs work with any language (Java, Python, C++, etc.). Tools differ but concepts are the same.

Q: Cost to implement? > A: Zero software costs (all open source). Investment is team training time (~2 weeks) and some CI/CD integration work.

Post-Demo Actions

Immediate Cleanup

```
# Stop load generator
if [ -f /tmp/load-gen.pid ]; then
  kill $(cat /tmp/load-gen.pid)
  rm /tmp/load-gen.pid
fi

# Verify all services recovered
docker compose ps

# Clean up any chaos experiment remnants
docker compose restart

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

Share Artifacts

Create a demo package:

```
cd ~/demo-artifacts-$(date +%Y%m%d)

# Create summary document
cat > DEMO_SUMMARY.md << 'EOF'
# SRE Demo - Chaos Engineering, Flamegraphs, and eBPF

## Date
$(date)

## Attendees
- [List attendees]

## What We Demonstrated
1. CPU flamegraph profiling (10 min)
2. Network partition chaos test (10 min)
3. CPU throttling chaos test (10 min)

## Key Findings
- Database operations consume 40% of CPU
- Network failure recovery time: 5.8 seconds
- System capacity: 3x current load before degradation
- Graceful degradation confirmed under CPU pressure

## Artifacts Included
- CPU flamegraphs (SVG files)
- Chaos experiment logs
- eBPF traces
- Grafana dashboard screenshots

## Next Steps
[Document action items from demo discussion]
EOF

# Compress for sharing
tar -czf ../sre-demo-$(date +%Y%m%d).tar.gz .
cd ..

echo "Demo package ready: sre-demo-$(date +%Y%m%d).tar.gz"
```

Follow-Up Email Template

```
Subject: SRE Demo Follow-up - Chaos Engineering & Observability Tools
```

```
Hi Team,

Thank you for attending today's demonstration of chaos engineering,
flamegraph profiling, and eBPF observability tools.

DEMO HIGHLIGHTS:
• Profiled live system with zero code changes
• Injected network failures and observed auto-recovery (5.8s)
• Measured system capacity (3x current load)
• Traced kernel behavior with eBPF

KEY FINDINGS:
• 40% of CPU time in database operations (optimization opportunity)
• Network partition recovery validated (retry logic working correctly)
• Graceful degradation under CPU pressure (no hard failures)
• System capacity well understood for scaling decisions

ARTIFACTS ATTACHED:
• Flamegraph visualizations
• Chaos experiment reports
• eBPF trace samples
• Demo guide document

NEXT STEPS:
1. Review flamegraphs for optimization opportunities (by [date])
2. Run remaining chaos experiments in staging (by [date])
3. Schedule team training on bpftrace (proposed: [date])
4. Integrate profiling into CI/CD pipeline (Q[X] goal)

RESOURCES:
• Demo guide: docs/SRE_DEMO_GUIDE.md
• Chaos experiments: chaos-experiments/README.md
• Flamegraph guide: docs/FLAMEGRAPH_GUIDE.md

Questions? Let's discuss in our next team meeting or reach out directly.

Best regards,
[Your name]
```

Troubleshooting

Services Not Starting

Symptom: `docker compose ps` shows services as "Exited" or "Restarting"

Diagnosis:

```
# Check logs for specific service
docker logs simulated-exchange-trading-api

# Check all services
docker compose logs --tail=50
```

Common Issues:

1. **Port conflicts**:

   ```
   # Check what's using ports
   sudo lsof -i :8080
   sudo lsof -i :5432

   # Kill conflicting process
   sudo kill -9 <PID>
   ```

2. **Database not ready**:

   ```
   # Wait for postgres health check
   docker compose up -d postgres
   sleep 30
   docker compose up -d
   ```

3. **Out of disk space**:

   ```
   df -h
   docker system prune -a  # Careful: removes unused images
   ```

Grafana Shows No Data

Symptom: Dashboards load but graphs are empty

Diagnosis:

```
# Check Prometheus is scraping
curl http://localhost:9090/api/v1/targets

# Should show all targets "up"
```

Fix:

```
# Restart Prometheus
docker compose restart prometheus

# Verify metrics endpoints
curl http://localhost:8080/metrics
curl http://localhost:8081/metrics
curl http://localhost:8082/metrics
```

```
# Check Prometheus config
docker exec simulated-exchange-prometheus cat /etc/prometheus/prometheus.yml
```

Flamegraph Generation Fails

Symptom: `make profile-cpu` fails with "connection refused"

Diagnosis:

```
# Check service is running
docker ps | grep trading-api

# Check pprof endpoint
curl http://localhost:8080/debug/pprof/
```

Fix:

```
# Ensure services fully started
docker compose restart trading-api
sleep 10

# Try manual profiling
./scripts/generate-flamegraph.sh cpu 30 8080
```

Chaos Experiment Hangs

Symptom: Chaos script runs but never completes

Diagnosis:

```
# Check if cleanup trap is working
ps aux | grep chaos

# Check Docker status
docker compose ps
```

Fix:

```
# Kill the script
pkill -f chaos-experiments

# Manually cleanup
docker compose restart
sudo iptables -F DOCKER  # Clear any iptables rules
docker update --cpus=2.0 --memory=2g simulated-exchange-trading-api
```

bpftrace Not Working

Symptom: "bpftrace: command not found" or permission errors

Diagnosis:

```
# Check if installed
which bpftrace

# Check kernel version
uname -r  # Need 4.9+

# Check permissions
sudo bpftrace --version
```

Fix:

```
# Install bpftrace
sudo apt-get install -y bpftrace  # Ubuntu/Debian
sudo dnf install -y bpftrace      # RHEL/Fedora

# If kernel too old
sudo apt-get install -y linux-headers-$(uname -r)

# Add user to tracing group (optional)
sudo usermod -aG tracing $USER
```

Load Generator Not Working

Symptom: No orders visible in Grafana after starting load generator

Diagnosis:

```
# Check if process running
ps aux | grep generate-load

# Check trading API
curl http://localhost:8080/health

# Check Prometheus query
curl -s --data-urlencode 'query=rate(orders_total[1m])' \
  http://localhost:9090/api/v1/query
```

Fix:

```
# Restart load generator
pkill -f generate-load
./generate-load.sh 3600 30 &

# Verify orders submitting
curl -X POST http://localhost:8080/orders \
  -H "Content-Type: application/json" \
  -d '{"symbol":"BTCUSD","type":"MARKET","side":"BUY","quantity":1.0}'
```

Appendix

A. Complete Make Target Reference

Profiling Targets:

```
make profile-cpu           # 30s CPU profile
make profile-cpu-long      # 60s CPU profile
make profile-heap          # Memory allocations
make profile-goroutine     # Concurrency state
make profile-allocs        # Allocation patterns
make profile-block         # Blocking operations
make profile-mutex         # Lock contention
make profile-all           # All of the above
make profile-view          # Open flamegraphs in browser
make profile-analyze       # AI analysis of latest profile
```

Chaos Targets:

```
make chaos-all             # Run all 10 experiments (60+ min)
make chaos-db-death        # Database sudden death
make chaos-network-partition # Network partition (eBPF)
make chaos-oom-kill        # Memory OOM kill
make chaos-cpu-throttle    # CPU throttling
make chaos-disk-io         # Disk I/O starvation (eBPF)
make chaos-conn-pool       # Connection pool exhaustion
make chaos-redis           # Redis cache failure
make chaos-nginx           # Nginx proxy failure
make chaos-latency         # Network latency injection
make chaos-cascade         # Multi-service cascade
make chaos-flamegraph      # Flamegraph during chaos
```

Utility Targets:

```
make help                  # Show all targets
make setup                 # Initialize project
make clean                 # Clean build artifacts
make profile-help          # Profiling documentation
```

B. Prometheus Query Examples

Order Rate:

```
rate(orders_total[1m])
```

Average Latency:

```
rate(order_processing_duration_seconds_sum[1m]) /
rate(order_processing_duration_seconds_count[1m])
```

Error Rate:

```
rate(orders_total{status="FAILED"}[1m]) /
rate(orders_total[1m])
```

95th Percentile Latency:

```
histogram_quantile(0.95,
  rate(order_processing_duration_seconds_bucket[1m]))
```

Database Connection Count:

```
database_connections_active
```

C. Grafana Dashboard URLs

After logging in (admin/admin123):

- **System Overview**: http://localhost:3000/d/system-overview
- **Trading API**: http://localhost:3000/d/trading-api
- **Market Simulator**: http://localhost:3000/d/market-simulator
- **Order Flow**: http://localhost:3000/d/order-flow
- **Database**: http://localhost:3000/d/database
- **Redis**: http://localhost:3000/d/redis

D. Useful bpftrace One-Liners

TCP connections:

```
sudo bpftrace -e 'kprobe:tcp_connect { printf("%s -> %s\n", comm, ntop(args->sk->__sk_commor
```

File opens:

```
sudo bpftrace -e 'tracepoint:syscalls:sys_enter_openat { printf("%s: %s\n", comm, str(args->
```

Process CPU time:

```
sudo bpftrace -e 'profile:hz:99 /comm == "trading-api"/ { @[ustack] = count(); }'
```

Memory allocations:

```
sudo bpftrace -e 'tracepoint:kmem:kmalloc { @[comm] = sum(args->bytes_alloc); }'
```

E. Docker Compose Quick Reference

Start all services:

```
docker compose -f docker-compose.yml -f docker/docker-compose.caddy.yml up -d
```

View logs:

```
docker compose logs -f trading-api
docker compose logs --tail=100
```

Restart service:

```
docker compose restart trading-api
```

Stop all:

```
docker compose down
```

Clean everything:

```
docker compose down -v  # Remove volumes too
docker system prune -a  # Clean unused images
```

F. Emergency Recovery

If demo environment is completely broken:

```
#!/bin/bash
# Nuclear option: Complete reset

# Stop everything
docker compose down -v

# Remove chaos artifacts
sudo iptables -F DOCKER
sudo tc qdisc del dev docker0 root 2>/dev/null || true

# Clean Docker
docker system prune -f

# Restart from scratch
docker compose -f docker-compose.yml -f docker/docker-compose.caddy.yml up -d

# Wait for initialization
sleep 60

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

echo "Environment reset complete"
```

Document Version History

Version	Date	Changes	Author
1.0	2025-11-15	Initial creation	[Your name]

License & Attribution

This demonstration guide is part of the Simulated Exchange SRE training program.

Tools Used: - Flamegraphs: Brendan Gregg (https://www.brendangregg.com/flamegraphs.html) - bpftrace: IO Visor Project (https://github.com/iovisor/bpftrace) - Prometheus: CNCF Project (https://prometheus.io/) - Grafana: Grafana Labs (https://grafana.com/)

End of Document