Interview Demo Script - AI Cost Optimization Dashboard

Duration: 5-7 minutes

Format: Live demo + technical deep dive

Goal: Show AI + DevOps + Business value skills

Pre-Demo Setup (Before Interview)

```bash
# Ensure everything works
cd ~/Desktop/PERSONAL_PROJECTS/AI-Cost-Optimization-Dashboard
source venv/bin/activate
python3 cost_optimizer.py  # Test run

# Have these ready:
# 1. Terminal with clean output
# 2. Browser tabs: GitHub repo, AWS Cost Explorer (same date range)
# 3. Code editor with cost_optimizer.py open
# 4. Screenshots folder visible
```

Demo Flow

Part 1: Problem Statement (60 seconds)

Say this:

> "I built this to solve a real problem I encountered: manual AWS cost analysis is time-consuming and often misses optimization opportunities.
> **The challenge**: AWS bills are complex—200+ services, nested pricing. FinOps teams spend hours reviewing spend, but generic alerts like 'EC2 costs increased' don't tell you *what* to do about it.
> **My solution**: An AI-powered dashboard that automates this entirely. It analyzes AWS spending, identifies specific optimizations with dollar savings, and ranks them by ROI. Let me show you."

Why this works: Shows business context, not just technical skills.

Part 2: Live Demo (2-3 minutes)

Step 1: Run the Script

```
bash

python3 cost_optimizer.py
```

As it runs, narrate:

> "First, it fetches 30 days of AWS Cost Explorer data using boto3. This is my personal test environment, about $6.62 in total spend."

Point to screen:

- ✅ Date range: 2026-01-09 to 2026-02-08
- ✅ Total cost: $6.62
- ✅ Services found: 15

Step 2: Visual Analysis

Point to the bar chart:

> "Notice the visual cost distribution. This ASCII bar chart makes patterns immediately visible. EKS is consuming 53.5% of my spend—that's the Kubernetes control plane."

Point to trend:

> "The trend analysis compares this period to the previous 30 days. If costs are growing, it flags it with an up arrow."

Why this matters: Visual communication of data, not just raw numbers.

Step 3: AI Recommendations

Scroll to recommendations section:

> "Now here's where AI adds value. Instead of generic advice like 'reduce costs,' Claude analyzes the data with a structured FinOps framework I designed."

Point to first recommendation (KMS Optimization):

> "Look at this format:
> - **What**: AWS KMS API calls
> - **Why**: $1.82/month (27.5%) is disproportionate for such low compute
> - **Action**: Specific steps—audit CloudTrail logs, consolidate keys
> - **Savings**: $1-1.50/month

> - **Risk**: Low
> - **Effort**: Medium (1-4 hours)
>
> This isn't just 'you're spending too much'—it's a work ticket ready to implement."

Point to ROI ranking:

> "Then it ranks all recommendations by ROI. 'Quick wins' with low risk come first. Architecture changes with high complexity come last. This helps teams prioritize."

Part 3: Technical Deep Dive (2-3 minutes)

Open code editor (cost_optimizer.py):

Show Prompt Engineering

Navigate to line ~260 (AI prompt):

> "The key to making AI useful is prompt engineering. Let me show you my FinOps prompt.
> [Scroll through prompt]
> Notice I don't just say 'analyze costs.' I require:
> - Specific format (What/Why/Action/Savings/Risk/Effort)
> - ROI prioritization
> - Constraints (only measurable $ impact, assume production environment)
>
> This structured approach is what makes the output actionable."

Why this matters: Shows you understand AI isn't magic—it's prompt design.

Show Error Handling

Navigate to line ~470 (low cost handling):

> "Production code needs edge case handling. For example, if costs are under $1, I provide context—could be free tier, testing, or new account—and ask if they want to continue.
> This graceful degradation prevents confusion when analyzing dev environments."

Why this matters: Shows production thinking, not just "happy path" code.

Show Trend Analysis

Navigate to line ~215 (previous period comparison):

> "For trend detection, I make a second Cost Explorer API call for the previous period—same number of days, offset by the analysis window.
> Then I calculate percent change and direction. This identifies cost growth early, before it becomes a budget problem."

Why this matters: Shows you think about business outcomes, not just technical implementation.

Part 4: Business Impact (1 minute)

Close the code, return to terminal:

> "Let me put this in business terms:
> **Time savings**: Manual cost analysis takes 4 hours/week. This script runs in 5 minutes. That's 95% time reduction—200 hours/year for a FinOps team.
> **Cost savings**: Even in my $6.62 test environment, it identified $3-5/month in optimizations. Scale that to a production environment spending $50K/month, and you're looking at $5K+/month in savings—$60K annually.
> **Risk reduction**: The risk assessment prevents teams from implementing high-risk changes without proper evaluation.
> This is infrastructure intelligence, not just automation."

Part 5: Q&A Preparation

Common Questions & Answers:

Q: "Why Claude instead of ChatGPT or cheaper models?"

A:

> "Claude excels at structured, context-aware analysis. I tested GPT-3.5 initially—it gave generic recommendations. Claude 3.5 Sonnet understands AWS service relationships and business context. Cost-wise, each analysis is ~$0.002—less than a penny. Running weekly = $0.40/month. Negligible vs the $5K+ it helps save."

Q: "How does this compare to AWS Trusted Advisor?"

A:

> "Trusted Advisor is great for compliance checks—security groups, IAM policies. But it's rule-based. This adds AI-powered business context. For example, Trusted Advisor might say 'low utilization RDS instance.' My tool would say: 'This RDS instance was last accessed 30 days ago during the Q4 promo—

> consider archiving post-campaign data to S3 and terminating the instance.'
> It's the *why* and *what to do* that make it actionable."

Q: "What was the hardest technical challenge?"

A:

> "Prompt engineering. My first version of the Claude prompt gave vague output like 'consider optimizing EC2 costs.'
> I iterated to a structured FinOps framework requiring specific details. The breakthrough was realizing I needed *constraints*—'only recommend changes with measurable $ impact' and 'assess risk level.'
> That's when recommendations became production-ready. It taught me that AI output quality is directly tied to prompt design quality."

Q: "How would you scale this for a large organization?"

A:

> "Great question. Current limitations:
>
> 1. Single AWS account (doesn't handle consolidated billing)
> 2. No historical tracking (can't measure actual savings over time)
> 3. Manual execution (should be automated weekly)
>
> **My roadmap**:
>
> - Multi-account support (analyze across dev/staging/prod)
> - Savings tracker (compare recommendations to actual spend reduction)
> - Web UI (make it self-service for all teams)
> - Jira integration (auto-create cost optimization tickets)
> - Terraform deployment (one-click setup for new accounts)
>
> I prioritized getting MVP working first, then iterate. That's how I approach production systems."

Q: "Show me the code quality—how do you handle errors?"

A: [Open code editor, show error handling sections]

> "I handle several edge cases:
>
> 1. **Missing AWS credentials**: Clear error message with remediation steps
> 2. **Cost Explorer not enabled**: Directs user to enable it
> 3. **Zero/low costs**: Provides context (free tier, testing) and prompts user

4. **API failures**: Graceful degradation with error logging
5. **DRY_RUN mode**: Allows testing without API charges

I also use try/except blocks around all API calls and provide user-friendly error messages, not stack traces."

Q: "What would you add if you had another week?"

A:

"Three features:

1. **Budget forecasting**: Use historical data + AI to predict next month's spend
2. **Anomaly detection**: Flag unusual spikes (e.g., 'Lambda costs 10x normal—possible runaway function')
3. **Auto-remediation**: For low-risk optimizations (e.g., delete old snapshots), auto-generate Terraform to implement changes

All three build on the foundation I've created—they're natural extensions of the cost analysis + AI recommendation pattern."

Q: "How do you test AI features?"

A:

"Good question. AI is non-deterministic, so traditional unit tests don't work well.
My approach:

1. **DRY_RUN mode**: Mock AI responses for fast iteration
2. **Regression testing**: Save known-good outputs, compare new runs
3. **Prompt versioning**: Track prompt changes in git, document why
4. **Real-world validation**: Run on actual AWS accounts, verify recommendations make sense
5. **Cost tracking**: Monitor Claude API usage to catch prompt bugs (e.g., infinite loops)

For this project, I'd add pytest tests for the data processing logic, and human review for AI output quality."

Closing Statement (30 seconds)

"This project demonstrates three things I bring to your team:

1. **AI integration skills**: I can leverage Claude, GPT, or other models to add intelligence to infrastructure
2. **Production mindset**: Error handling, risk assessment, documentation—not just proof-of-concepts

> 3. **Business focus**: Every feature is tied to measurable value—time saved, money saved
>
> I built this in 3 days as part of my AI-powered DevOps portfolio. I'm excited to bring this approach to [Company Name]'s infrastructure challenges."

Post-Demo: Share Resources

Have these links ready in chat:

```
GitHub: https://github.com/yourusername/ai-cost-optimization-dashboard
Live Demo: [Video link if recorded]
Case Study: [Link to CASE_STUDY.md in repo]
```

Demo Checklist ✅

Before Interview:

- [ ] Test run works without errors
- [ ] Screenshots are saved and accessible
- [ ] GitHub repo is public and README is polished
- [ ] Browser tabs ready (GitHub, AWS Console)
- [ ] Code editor open to key sections
- [ ] Practiced demo at least 2x (timing!)

During Demo:

- [ ] Start with problem statement (business context)
- [ ] Show live execution (not just screenshots)
- [ ] Point out specific features (chart, trend, recommendations)
- [ ] Dive into code (prompt engineering, error handling)
- [ ] Close with business impact (time saved, $ saved)
- [ ] Handle questions confidently (use Q&A prep above)

After Demo:

- [ ] Share GitHub link in chat
- [ ] Offer to send case study document
- [ ] Follow up with thank-you email including demo recording

---

Time Breakdown

Total: 5-7 minutes

- Problem statement: 60s
- Live demo: 2-3 min
- Technical deep dive: 2-3 min
- Business impact: 60s
- Q&A: Variable (have answers ready)

Practice this! Time yourself. 7 minutes is ideal—shows depth without losing attention.

Pro Tips

1. **Start strong**: First 30 seconds set the tone. Lead with business problem, not tech stack.
2. **Show, don't tell**: Run the script live. Even if you have screenshots, seeing it execute is more impressive.
3. **Have a narrative**: "Problem → Solution → Impact" flow keeps audience engaged.
4. **Anticipate questions**: The Q&A prep above covers 80% of what interviewers ask.
5. **Link to next project**: "This is part of my AI-powered DevOps portfolio. Next, I'm building a Terraform AI generator." Shows forward momentum.
6. **Confidence, not arrogance**: "I'm proud of this" not "This is the best cost optimizer ever built."

You've got this! Practice the demo 2-3 times, and you'll nail it. 🚀