CS 470 Final Reflection

Student Name: Valerie Dawson
Date: October 21st, 2025
Presentation Link: https://youtu.be/SRmySti7x6k?si=eMVMhTmZc1H6WT1r

Experiences and Strengths

Professional Growth and Marketable Skills

Completing CS 470 has been a transformative capstone experience as I prepare to transition from academia into a professional career in cloud engineering. This course has bridged the gap between theoretical computer science knowledge and practical cloud-native application development, providing me with highly marketable skills that align directly with my career aspirations in cloud engineering and full-stack development.

Throughout this course, I have developed and mastered several critical competencies that make me a competitive candidate in today's technology job market:

Containerization and Orchestration: I gained hands-on experience with Docker, creating multi-stage Dockerfiles and implementing Docker Compose configurations for the lafs-web (Learn Angular From Scratch) application. This practical experience taught me how to create reproducible development environments, manage dependencies effectively, and prepare applications for deployment at scale. Understanding containerization is fundamental for modern DevOps practices and is a skill highly valued by organizations transitioning to cloud-native architectures.

AWS Cloud Services Integration: Building upon my previous AWS coursework, I deepened my expertise by implementing a complete serverless architecture using multiple AWS services. I successfully integrated DynamoDB for NoSQL database management, configured API Gateway for RESTful API endpoints, implemented AWS Lambda functions for serverless compute, and utilized S3 for static asset storage. This multi-service integration taught me how different cloud components work together to create scalable, resilient applications.

Full-Stack Development with Modern Frameworks: The hands-on implementation of the 1 MEAN stack (MongoDB/DynamoDB, Express, Angular, Node.js) reinforced my ability to work across the entire application stack. Configuring Angular Material components, managing TypeScript configurations, and implementing responsive front-end interfaces has strengthened my versatility as a developer

Infrastructure as Code and Configuration Management: Through creating Docker configurations, managing environment variables, and documenting deployment processes, I learned the importance of treating infrastructure as code. This approach ensures consistency, enables version control for infrastructure, and facilitates collaboration in team environments.

API Development and Testing: Completing the API implementation and conducting thorough testing taught me the critical importance of creating robust, well-documented interfaces. I learned to implement proper error handling, design RESTful endpoints following industry best practices, and validate functionality through systematic testing methodologies.

CS 470 Final Reflection

Student Name: Valerie Dawson
Date: October 21st, 2025
Presentation Link: https://youtu.be/SRmySti7x6k?si=eMVMhTmZc1H6WT1r
These skills complement my existing foundation from my AAS in IT and the courses I completed while obtaining BS in Computer Science, where I developed proficiency in Java, C++, and Python, gained experience with both Waterfall and Agile Scrum methodologies, and completed diverse projects including a hospital management system, pet shelter dashboard, UI/UX design for AgriPulse, 3D scene rendering with OpenGL, and AI exploration projects.

My Strengths as a Software Developer

As I reflect on my journey through computer science education and this capstone course, several key strengths emerge that define my capabilities as a software developer:

Full-Stack Versatility: My ability to work confidently across the entire technology stack, from database design through API development to front-end implementation, positions me as a valuable team member who can contribute to any layer of application architecture. Whether optimizing database queries in DynamoDB, implementing Lambda functions, or crafting responsive Angular interfaces, I can navigate the complete development lifecycle.

Problem-Solving and Debugging: Throughout my coursework, particularly in CS 470, I developed resilience in troubleshooting complex integration issues. Containerization challenges, API configuration problems, and cloud service integration issues required systematic debugging approaches and creative problem-solving skills that are essential in professional development environments.

Adaptability and Continuous Learning: The rapid evolution of cloud technologies demands continuous learning. My ability to quickly absorb new concepts, demonstrated by mastering containerization and serverless architectures in a single semester while balancing multiple courses, shows my commitment to staying current with emerging technologies.

Systems Thinking: My diverse project experience has developed my ability to see the bigger picture and understanding how individual components interact within larger systems. This holistic perspective is crucial for cloud architecture, where decisions about one service can have cascading effects throughout the application.

Documentation and Communication: Creating presentations, maintaining GitHub portfolios, and documenting technical processes throughout CS 470 has strengthened my ability to communicate complex technical concepts clearly. This skill is invaluable for collaborating with both technical and non-technical stakeholders.

Roles I am Prepared to Assume

Based on my comprehensive skill set and experiences, I am well-prepared to assume several roles in professional environments:

CS 470 Final Reflection

Student Name: Valerie Dawson
Date: October 21st, 2025
Presentation Link: https://youtu.be/SRmySti7x6k?si=eMVMhTmZc1H6WT1r
Cloud Engineer/Cloud Developer: My hands-on experience with AWS services, containerization, and serverless architectures prepares me to design, implement, and maintain cloud-based solutions. I can architect scalable applications, optimize cloud resource utilization, and implement best practices for security and performance.

Full-Stack Developer: With demonstrated proficiency across front-end frameworks (Angular), back-end technologies (Node.js, Express), database systems (DynamoDB, experience with SQL), and cloud infrastructure, I can contribute to all aspects of web application development.

DevOps Engineer: My experience with Docker, infrastructure as code principles, and CI/CD concepts positions me to bridge the gap between development and operations teams. I understand the importance of automation, continuous integration, and maintaining reliable deployment pipelines.

Junior Cloud Architect: While I recognize this role typically requires additional professional experience, my educational foundation provides a strong starting point. I understand how to 3 evaluate trade-offs between different cloud services, design for scalability and resilience, and make informed architectural decisions.

Software Developer (Entry-Level): My broad technical foundation, including multiple programming languages (Java, C++, Python, JavaScript/TypeScript), version control proficiency, and experience with both Waterfall and Agile methodologies, prepares me to contribute effectively to software development teams across various domains. I am particularly excited about opportunities that allow me to work with cloud technologies while continuing to expand my expertise across the full development stack. My goal is to join an organization where I can apply my current skills while learning from experienced professionals and growing into more senior cloud engineering roles.

Planning for Growth

Microservices and Serverless Architectures for Future Efficiency
As the lafs-web application evolves and user demand increases, transitioning from a monolithic architecture to microservices or expanding serverless implementations offers significant advantages for management efficiency and scale. Based on my experience in CS 470 and understanding of cloud-native design patterns, I can identify several strategic approaches for future growth.

Microservices Decomposition Strategy: The current lafs-web application could be decomposed into distinct microservices organized around business capabilities. For example, user authentication and authorization could become an independent microservice, course content management could be separated into its own service, and user progress tracking could be isolated as another service. Each microservice would maintain its own DynamoDB table or database instance, communicate through API Gateway endpoints, and scale independently based on demand patterns. This decomposition provides several efficiencies. First, different services can be scaled independently if authentication requests surge during peak enrollment periods, but content

CS 470 Final Reflection

Student Name: Valerie Dawson
Date: October 21st, 2025
Presentation Link: https://youtu.be/SRmySti7x6k?si=eMVMhTmZc1H6WT1r
delivery remains stable, only the authentication microservice needs additional resources. Second, development teams can work on different services simultaneously without creating deployment bottlenecks, accelerating 4 feature delivery. Third, technology choices can be optimized per service, perhaps authentication uses Lambda with Python for its robust security libraries, while content delivery uses Node.js for its performance characteristics.

Serverless Architecture Expansion:

The application already utilizes AWS Lambda for certain functions, but a more comprehensive serverless approach could yield additional benefits. API Gateway integration with Lambda provides automatic scaling without managing server infrastructure. By implementing Lambda functions for all backend logic, the application achieves true pay-per-execution pricing and eliminates idle resource costs.

For the lafs-web application specifically, serverless functions could handle user registration, course enrollment processing, progress tracking updates, and content recommendations. AWS Step Functions could orchestrate complex workflows, such as the multi-step process of enrolling a user, provisioning course materials, and sending confirmation notifications. DynamoDB Streams could trigger Lambda functions automatically when data changes, enabling real-time features like progress notifications without polling overhead.

Handling Scale Through Cloud-Native Patterns:

Effective scaling requires implementing several key strategies:

Horizontal Scaling: Containerized microservices deployed through Amazon ECS (Elastic Container Service) or EKS (Elastic Kubernetes Service) can automatically scale container instances based on CPU utilization, memory consumption, or custom CloudWatch metrics. Application Load Balancers distribute traffic across healthy instances, ensuring high availability.

Caching Layers: Implementing Amazon ElastiCache (Redis or Memcached) reduces database load by caching frequently accessed data like course catalogs, user profiles, and static content. CloudFront CDN caches static assets at edge locations, reducing latency for global users and decreasing origin server load.

Database Scaling: DynamoDB's on-demand capacity mode automatically scales read and write throughput based on traffic patterns. For complex query patterns, DynamoDB Global Secondary Indexes can be strategically designed to support specific access patterns without impacting primary table performance. For scenarios requiring cross-region redundancy, DynamoDB Global Tables 5 provide multi-region replication with automatic conflict resolution.

Asynchronous Processing: Amazon SQS (Simple Queue Service) can decouple components, allowing services to communicate asynchronously. For example, when users upload profile images, the upload service places a message in SQS, and a separate processing service resizes and

CS 470 Final Reflection

Student Name: Valerie Dawson
Date: October 21st, 2025
Presentation Link: https://youtu.be/SRmySti7x6k?si=eMVMhTmZc1H6WT1r
optimizes images without blocking the user's request. This pattern improves perceived performance and enables graceful degradation under heavy load.

Comprehensive Error Handling Strategy:

Robust error handling is crucial for production applications at scale:

Circuit Breaker Pattern: Implementing circuit breakers prevents cascading failures when dependent services experience issues. If the authentication service becomes unresponsive, the circuit breaker opens after a threshold of failures, immediately returning cached responses or graceful error messages rather than waiting for timeouts. This protects the overall system health.

Retry Logic with Exponential Backoff: Transient failures in distributed systems are inevitable. Implementing intelligent retry logic with exponential backoff and jitter prevents overwhelming struggling services while automatically recovering from temporary issues. AWS SDK for Lambda includes built-in retry mechanisms that can be customized for specific use cases.

Dead Letter Queues: Messages that fail processing repeatedly can be routed to Dead Letter Queues (DLQs) for later analysis without blocking the main processing pipeline. This enables developers to investigate problematic requests without losing data or disrupting service.

Distributed Tracing and Monitoring: AWS X-Ray provides distributed tracing across microservices, enabling visualization of request flows and identification of performance bottlenecks. CloudWatch Logs Insights allows querying logs across all services to correlate errors and understand system behavior during incidents. Setting up comprehensive CloudWatch Alarms ensures proactive notification of anomalies before they impact users.

Graceful Degradation: Services should be designed to function at a reduced capacity rather than failing completely. For example, if the recommendation engine fails, the application should still serve course content without personalized suggestions, perhaps falling back to popular courses or recently added content.

Cost Prediction and Management

Accurate cost prediction requires understanding of both the pricing models and expected usage patterns for each service:

Serverless Lambda Costs: Lambda pricing includes two components, compute time charged per millisecond at rates determined by allocated memory (ranging from 128MB to 10GB), and request charges ($0.20 per 1 million requests). For the lafs-web application, if an average API request executes a Lambda function with 512MB memory for 200ms, and the application serves 10 million requests monthly, the calculation would be approximately:- Compute: 10M requests × 200ms × 512MB memory unit = $8.33- Requests: 10M requests × $0.20/1M = $2.00- Total: ~$10.33 monthly (plus 1M requests free tier)

CS 470 Final Reflection

Student Name: Valerie Dawson
Date: October 21st, 2025
Presentation Link: https://youtu.be/SRmySti7x6k?si=eMVMhTmZc1H6WT1r
DynamoDB Costs: On-demand capacity mode charges $1.25 per million write requests and $0.25 per million read requests, plus $0.25 per GB storage monthly. Provisioned capacity mode offers predictable pricing when usage patterns are consistent, charging for reserved read/write capacity units. For applications with variable traffic like educational platforms (high during business hours, low overnight), on-demand mode typically provides better value and eliminates over-provisioning.

Container Costs (ECS/Fargate): Fargate pricing is based on vCPU-hours and GB-hours allocated to containers. A container with 0.5 vCPU and 1GB memory running continuously costs approximately $14.96 monthly per instance. However, containers run 24/7 regardless of traffic, unlike Lambda's per-execution billing.

Data Transfer and Storage: S3 storage costs $0.023 per GB for standard storage class, with additional charges for requests and data transfer out to the internet. CloudFront reduces these costs by caching content at edge locations, charging $0.085 per GB for data transfer out (varies by region), which is often less than direct S3 egress costs.

Cost Predictability: Containers vs. Serverless:

Containers generally provide more cost predictability but less cost efficiency for variable workloads:

Container Cost Predictability (Higher): When running ECS with Fargate or EKS, costs are based on provisioned resources that run continuously. If you provision three containers running 24/7, you can calculate exact monthly costs: 3 containers × 720 hours × vCPU rate + GB rate. This predictability simplifies budgeting but results in paying for idle capacity during low-traffic periods.

For example, if the lafs-web application requires three Fargate containers (0.5 vCPU, 1GB each) running continuously:- Monthly cost: 3 × $14.96 = ~$44.88 (predictable but constant)

Serverless Cost Predictability (Lower but More Efficient): Lambda costs fluctuate directly with usage, making precise prediction challenging without historical data. However, serverless architectures align costs with actual usage, eliminating idle resource expenses. During low-traffic overnight hours, Lambda costs approach zero, while container costs remain constant.

Using the same workload during peak hours but with 80% reduction overnight:- Peak hours (12hrs): High Lambda costs- Off-peak (12hrs): Minimal Lambda costs- Average: Potentially 40-60% cost reduction compared to 24/7 containers.

The most cost-predictable approach combines both: Use containers for consistent, baseline traffic and Lambda for variable or spiky workloads. This hybrid architecture provides budget predictability while capturing the efficiency benefits of serverless unpredictable components.

Pros and Cons for Expansion Decisions

CS 470 Final Reflection

Student Name: Valerie Dawson
Date: October 21st, 2025
Presentation Link: https://youtu.be/SRmySti7x6k?si=eMVMhTmZc1H6WT1r

Microservices Architecture:

Pros:

- **Independent Scalability:** Each service scales based on its specific demand patterns, optimizing resource utilization and costs.
- **Technology Flexibility:** Different services can use optimal technology stacks such as Python for data processing, Node.js for real-time features, Java for transaction-heavy components.
- **Fault Isolation:** Failures in one service don't cascade to the entire application; the course content service can continue functioning even if the recommendation engine fails.
- **Parallel Development:** Multiple teams can develop, test, and deploy services independently, accelerating feature delivery and reducing deployment risks.
- **Optimized Database Design:** Each service can have purpose-built data stores optimized for its access patterns, improving performance.

Cons:

- **Increased Complexity:** Managing dozens of microservices requires sophisticated orchestration, service discovery, and inter-service communication patterns.
- **Distributed System Challenges:** Network latency, partial failures, and eventual consistency issues become more prevalent and require careful design.
- **Operational Overhead:** More services mean more deployment pipelines, monitoring dashboards, logging configurations, and potential failure points to manage.
- **Testing Complexity:** Integration testing becomes more complex when services depend on each 8 other, requiring comprehensive test strategies and potentially expensive test environments.
- **Data Consistency**: Maintaining consistency across service boundaries requires implementing distributed transaction patterns like saga or event sourcing.

Serverless Architecture:

Pros:

- **Zero Server Management:** No infrastructure provisioning, patching, or capacity planning required, AWS handles all operational aspects
- **Automatic Scaling:** Lambda scales from zero to thousands of concurrent executions automatically without configuration.
- **Cost Efficiency for Variable Loads:** Pay only for actual execution time; no charges during idle periods, making it ideal for applications with unpredictable or spiky traffic patterns.
- **Built-in High Availability:** Lambda functions automatically run across multiple availability zones without additional configuration.
- **Rapid Development:** Focus on business logic rather than infrastructure management accelerates development cycles

CS 470 Final Reflection

Student Name: Valerie Dawson
Date: October 21st, 2025
Presentation Link: https://youtu.be/SRmySti7x6k?si=eMVMhTmZc1H6WT1r

Cons:

- **Cold Start Latency:** Lambda functions experience initialization delays (100-1000ms) when first invoked after idle periods, potentially impacting user experience for latency-sensitive operations.
- **Execution Time Limits:** Lambda functions have a maximum execution duration of 15 minutes, making them unsuitable for long-running processes
- **Vendor Lock-in**: Heavy reliance on AWS-specific services (Lambda, API Gateway, DynamoDB) makes migration to other cloud providers challenging.
- **Debugging Complexity:** Troubleshooting distributed serverless applications requires specialized tools and approaches compared to traditional monolithic debugging.
- **Cost Unpredictability:** While efficient for variable loads, extremely high traffic can result in unexpected costs without proper monitoring and alert

Container- Based Architecture (ECS/EKS):

Pros:

- **Consistent Environment**: Containers ensure applications run identically across development, testing, and production environments.
- **Control and Flexibility**: Greater control over runtime environment, operating system, and networking compared to serverless.
- Suitable for Stateful Applications: Better support for applications requiring persistent connections or long-running processes.
- **Predictable Costs**: Fixed hourly pricing for provisioned containers simplifies budget forecasting.
- **Portability**: Containers can run on any Kubernetes cluster or container orchestration platform, reducing vendor lock-in.

Cons:

- **Infrastructure Management:** Even with managed services like ECS/EKS, you're 9 responsible for container orchestration, scaling policies, and cluster health**.**
- **Idle Resource Costs:** Containers run continuously, consuming resources and incurring costs even during low-traffic periods.
- **Manual Scaling Configuration:** Requires defining scaling policies and metrics, with potential for over-provisioning or under-provisioning
- **Slower Scaling Response:** Scaling new container instances takes minutes compared to serverless milliseconds, potentially impacting sudden traffic spikes.

Elasticity and Pay-for-Service in Growth Planning

The Role of Elasticity:

CS 470 Final Reflection

Student Name: Valerie Dawson
Date: October 21st, 2025
Presentation Link: https://youtu.be/SRmySti7x6k?si=eMVMhTmZc1H6WT1r
Elasticity is the ability to dynamically acquire and release resources based on demand—is fundamental to cost-effective cloud applications. For the lafs-web application, elasticity enables matching resource allocation to actual user demand patterns rather than provisioning for peak capacity.

Educational platforms experience predictable usage patterns: high traffic during business hours and academic terms, lower traffic evenings and weekends, minimal traffic during holidays. Without elasticity, the application would need to maintain infrastructure sized for peak usage 24/7, resulting in 60-70% idle capacity costs. With elasticity, resources automatically scale down during low-traffic periods, potentially reducing infrastructure costs by 40-60%.

Implementation Strategies:

Application Load Balancer with Auto Scaling: ECS services or EC2 instances behind an Application Load Balancer can implement target tracking scaling policies. When average CPU utilization exceeds 70% for five minutes, additional instances launch automatically. When utilization drops below 40%, excess instances terminate. This maintains performance during traffic surges while minimizing costs during quiet periods.

Lambda Automatic Concurrency: Lambda's inherent elasticity requires no configuration, it automatically scales to handle concurrent requests up to account limits (default 1,000 concurrent executions, increasable to tens of thousands). This native elasticity makes serverless ideal for unpredictable workloads like new course launches or viral social media mentions driving traffic spikes.

DynamoDB Auto Scaling: DynamoDB's on-demand capacity mode provides elasticity without configuration, automatically accommodating throughput variations. Alternatively, provisioned capacity with auto scaling adjusts read/write capacity units based on utilization metrics, providing 10 cost advantages for consistent workloads while maintaining elasticity for variations.

Pay-for-Service Economic Model:

The pay-for-service model fundamentally changes financial planning for application growth:

Traditional Capital Expenditure: On-premises infrastructure requires upfront capital investment in servers, storage, and networking equipment, plus ongoing operational

CS 470 Final Reflection

Student Name: Valerie Dawson
Date: October 21st, 2025
Presentation Link: https://youtu.be/SRmySti7x6k?si=eMVMhTmZc1H6WT1r
expenses for power, cooling, and maintenance. Capacity must be provisioned for projected peak load, resulting in significant over-provisioning and sunk costs.

Cloud Operating Expenditure: Cloud services shift spending from capital expenditure to operational expenditure, converting fixed costs into variable costs aligned with usage. This has several strategic advantages:

Reduced Financial Risk: Launching new features or entering new markets no longer requires significant infrastructure investment. If a new course category attracts fewer students than projected, costs automatically adjust downward. This reduces the financial risk of innovation and experimentation.

***Improved Cash Flow*:** Rather than large upfront investments depreciating over years, cloud spending aligns with revenue generation. As the application gains users and generates revenue, infrastructure costs scale proportionally.

Optimization Opportunities: Granular usage-based pricing enables continuous cost optimization. Analyzing CloudWatch metrics might reveal that shifting batch processing jobs from Lambda to cheaper EC2 Spot instances reduces costs by 40% without impacting performance. Traditional infrastructure provides no such optimization flexibility

Optimization Opportunities: Granular usage-based pricing enables continuous cost optimization. Analyzing CloudWatch metrics might reveal that shifting batch processing jobs from Lambda to cheaper EC2 Spot instances reduces costs by 40% without impacting performance. Traditional infrastructure provides no such optimization flexibility.

Strategic Decision-Making Framework:

When planning for growth, decision-making should balance several factors:

For Predictable, Consistent Workloads: Container-based architectures with provisioned capacity provide cost predictability and operational simplicity. Core application services handling baseline user traffic are ideal candidates.

For Variable, Event-Driven Workloads: Serverless architectures maximize cost efficiency and minimize operational overhead. Background processing, API endpoints, and integration services 11 benefit from serverless patterns.

For Rapid Growth Scenarios: Prioritize elasticity and automation. As user base grows 10× or 100×, manually managing infrastructure becomes impossible. Services that

CS 470 Final Reflection

Student Name: Valerie Dawson
Date: October 21st, 2025
Presentation Link: https://youtu.be/SRmySti7x6k?si=eMVMhTmZc1H6WT1r
automatically scale without intervention (Lambda, DynamoDB on-demand, API Gateway) reduce operational burden during critical growth phases.

For Long-Term Cost Optimization: Implement FinOps practices to continuously monitor costs, analyze usage patterns, and optimize resource allocation. AWS Cost Explorer and CloudWatch Metrics provide data-driven insights for optimization decisions. Consider Reserved Instances or Savings Plans for predictable baseline workloads to reduce costs by 30-50% compared to on-demand pricing

Conclusion

Completing CS 470 has equipped me with practical cloud engineering skills that complement my comprehensive computer science education and diverse project experience. I have mastered containerization with Docker, implemented serverless architectures with AWS, and developed a deep understanding of how to architect applications for scale, resilience, and cost-efficiency.

As I begin my professional career in cloud engineering, I am confident in my ability to contribute immediately while continuing to grow. The technical skills, problem-solving capabilities, and systems thinking developed throughout my education position me to tackle real-world challenges in cloud-native application development.

The future of the lafs-web application–and applications like it, lies in leveraging microservices, serverless architectures, and cloud-native design patterns to achieve elasticity, cost efficiency, and operational excellence. With the foundation established in CS 470, I am prepared to design and implement these architectures, making strategic decisions that balance technical requirements, business objectives, and financial constraints.

I look forward to applying these skills in professional environments, contributing to innovative projects, and continuing my journey as a cloud engineer building scalable, efficient, and reliable applications in the cloud.