AWS DRS Orchestration Solution
Executive Prototype Summary

Minimum Viable Product

November 2025

Notices

Table of Contents

Abstract

This document provides executive leadership with a comprehensive assessment of the AWS DRS Orchestration Solution prototype, including current capabilities, demonstrated feasibility, critical gaps, and recommended path forward. The prototype successfully demonstrates that Competitor's Site Recovery Manager-like orchestration capabilities can be built on AWS Elastic Disaster Recovery Service (DRS), addressing the fundamental gap where AWS DRS provides replication but no enterprise-grade orchestration or automation.

This assessment covers prototype achievements, monthly operating costs ($12-40), significant work remaining in recovery plan execution testing and wave orchestration validation, and decision points for project continuation.

Target audience includes executive leadership, senior management, and technical decision-makers who need to evaluate the prototype's viability, understand the technical and business case for AWS DRS orchestration automation, and make informed decisions about continued investment and development priorities.

Introduction

When you evaluate disaster recovery solutions for AWS workloads, you encounter a critical architectural gap: AWS Elastic Disaster Recovery (DRS) provides excellent continuous replication capabilities but lacks the enterprise-grade orchestration and automation that organizations moving from VMware Site Recovery Manager, Zerto and even Azure Site Recovery expect. This gap creates operational challenges at scale - managing hundreds of servers, coordinating multi-tier application recovery, maintaining boot order dependencies, and executing repeatable disaster recovery procedures all require manual processes that are error-prone, time-consuming, and difficult to validate.

You need to understand whether building automated orchestration on top of AWS DRS is technically feasible, what has been demonstrated in this prototype phase, what critical work remains, and what investment would be required to reach operational readiness. This document provides that understanding through honest assessment of prototype achievements, transparent disclosure of gaps and limitations, realistic cost projections, and clear recommendations for next steps.

The document structure guides you through five major sections: the business problem and technical gap, prototype solution overview, current status with validated capabilities, critical work remaining with effort estimates, and cost analysis with decision recommendations. You'll find feature comparison tables showing VMware SRM capabilities versus AWS DRS gaps, architectural diagrams illustrating the serverless solution design, and realistic timelines for reaching operational readiness. Supporting appendices provide technical depth including monthly cost breakdowns, feature parity assessments, and risk considerations.

Executive Summary

This prototype assessment establishes the technical feasibility and business case for AWS DRS Orchestration Solution, addressing the critical gap where AWS Elastic Disaster Recovery provides replication but no enterprise orchestration capabilities. The framework demonstrates proof-of-concept for automated multi-tier application recovery, visual server management, and API-driven disaster recovery execution - capabilities organizations expect from VMware Site Recovery Manager but which AWS DRS completely lacks.

Key Technical Achievement: The prototype successfully integrates 12 AWS services (DynamoDB, Lambda, Step Functions, API Gateway, Cognito, S3, CloudFront, Systems Manager, SNS, CloudFormation, IAM, DRS) into a cohesive serverless orchestration layer. Protection Groups with visual server selection work and have been tested. Server discovery automatically detects DRS source servers. The modern React UI provides VMware SRM-like user experience. API infrastructure handles authentication and CRUD (create, read, update, and delete) operations.

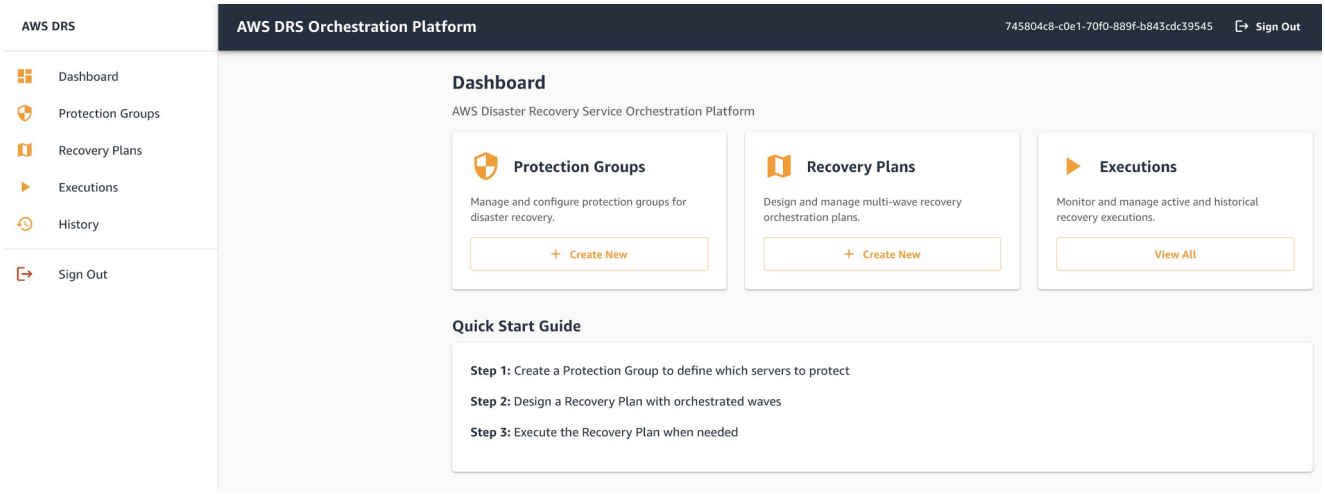
Critical Gap Assessment: AWS DRS replicates data continuously but provides zero orchestration capabilities. Managing 100+ servers manually without automated grouping, boot order control, dependency management, or recovery plan execution is operationally unacceptable at enterprise scale. This prototype proves automated orchestration is technically feasible using AWS-native services. The ConflictException fix represents major progress, but comprehensive validation testing remains essential before the solution can be trusted for actual disaster recovery operations.

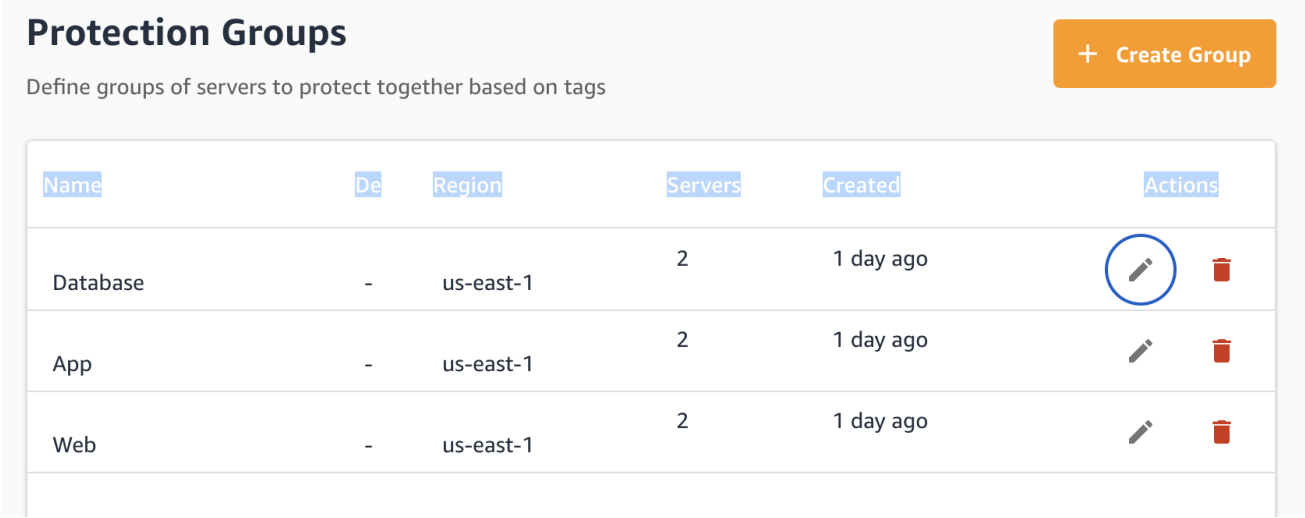
Work Remaining: Two major phases required –

- **Validation & Testing: 2-3 weeks intensive testing** of deployed ConflictException handling, recovery plan execution, wave orchestration, and failure scenarios;
- Core Missing Features: 6-8 weeks developing pre/post recovery scripts, network isolation, and enhanced monitoring;
- Operational Readiness: 4-6 weeks for comprehensive monitoring, security hardening, and documentation.
- **Total estimate: 2-3 months focused development** to reach operational confidence for disaster recovery use

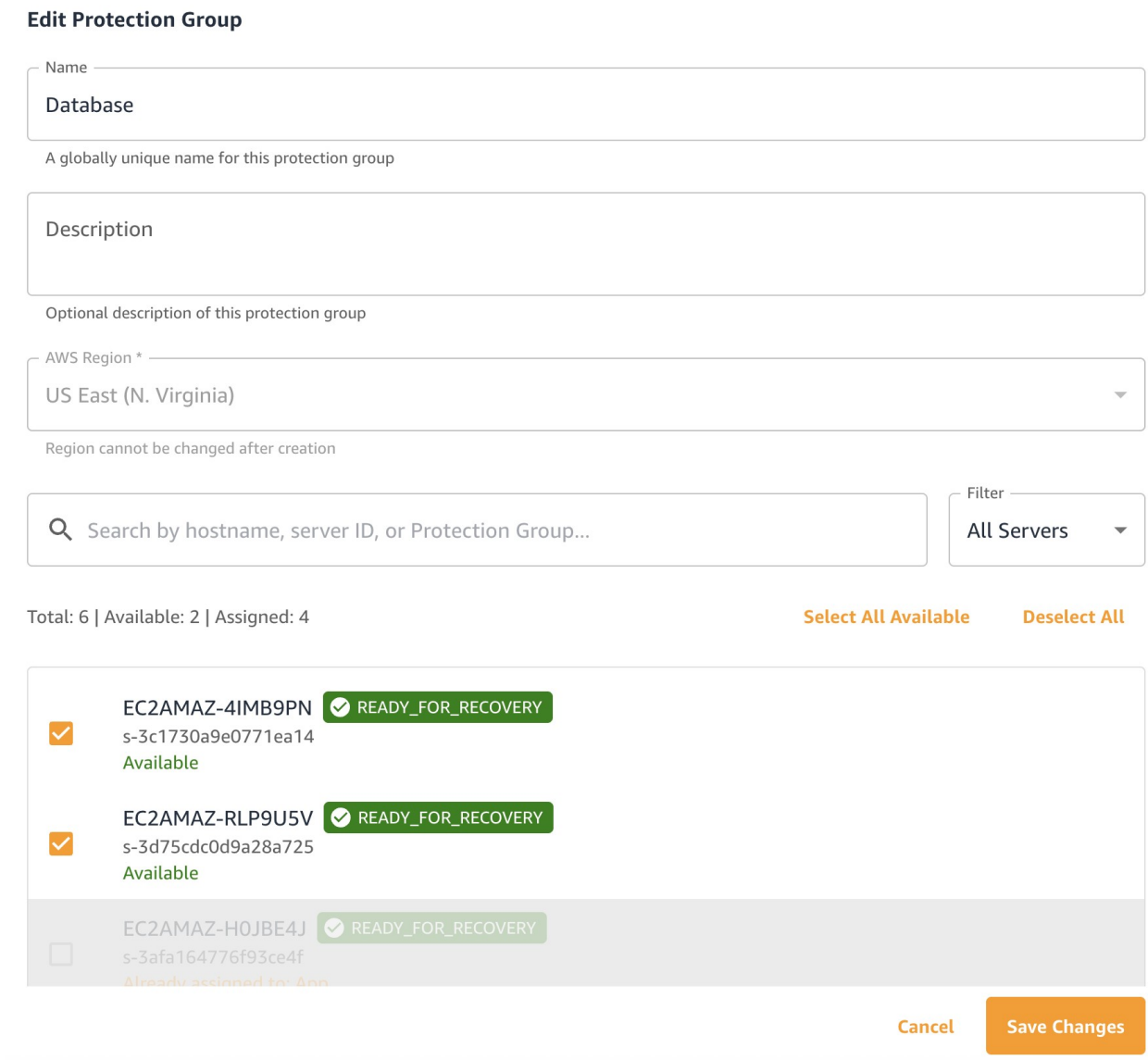
Key Features Demonstrated:

- **Protection Groups Management**: Visual server selection interface operational (VMware SRM-style)

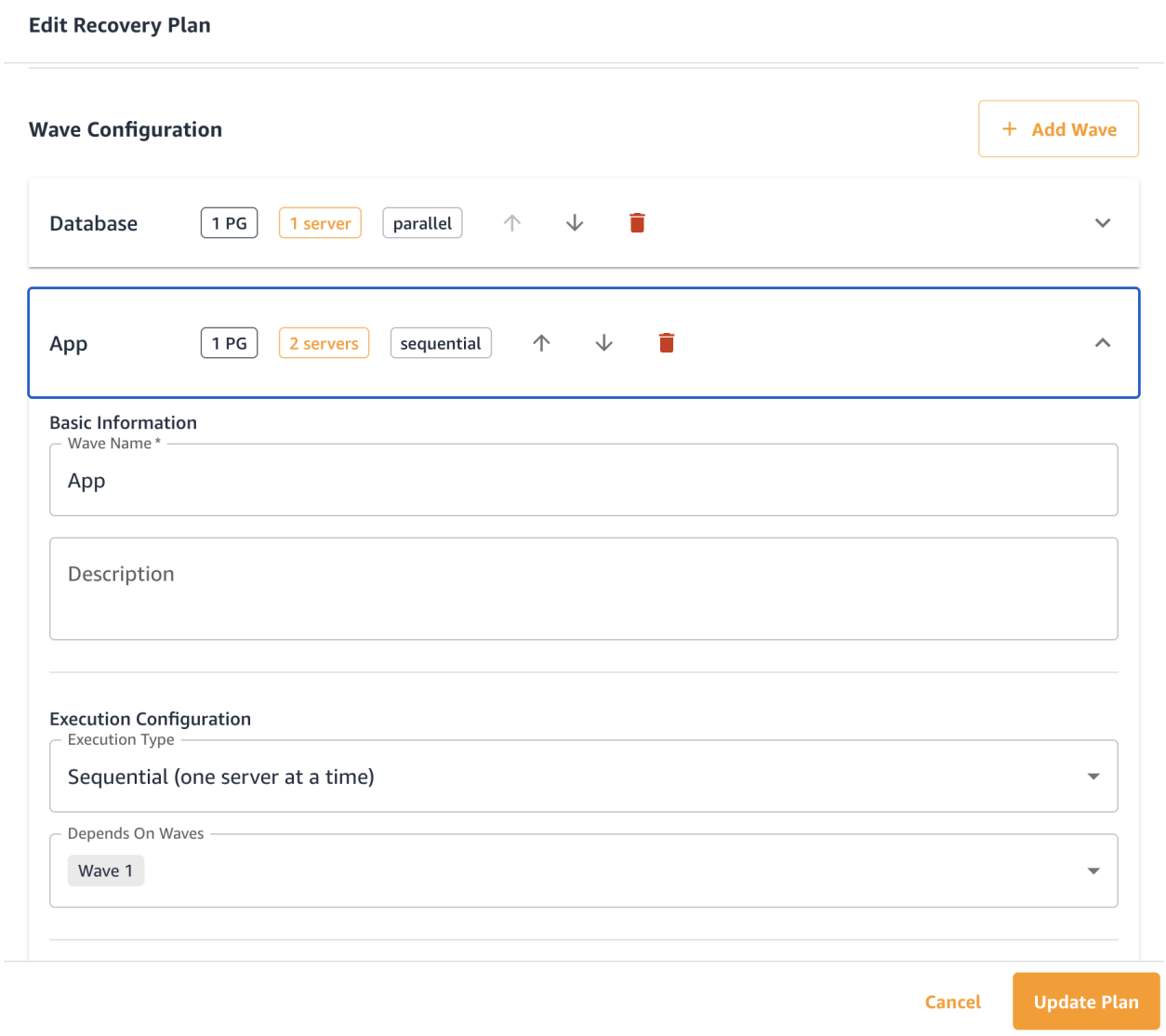




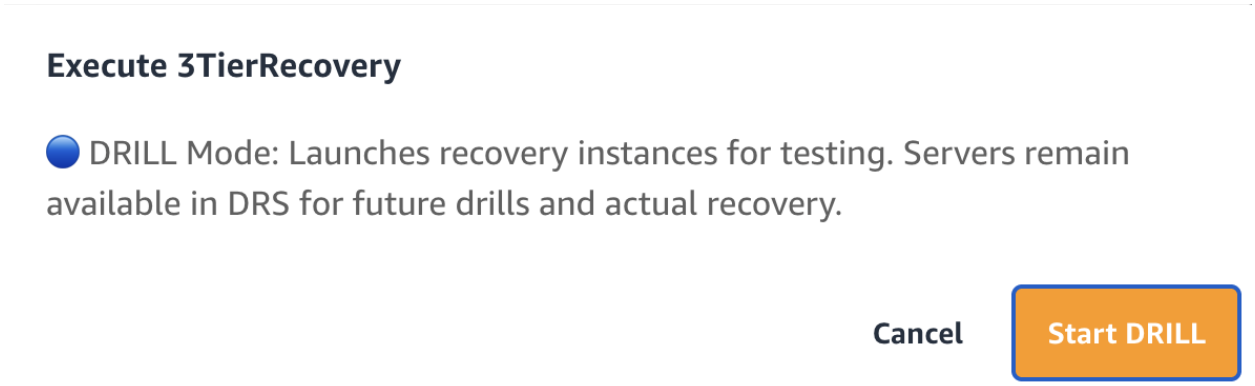
- **Automatic Server Discovery**: DRS source server detection working across AWS regions

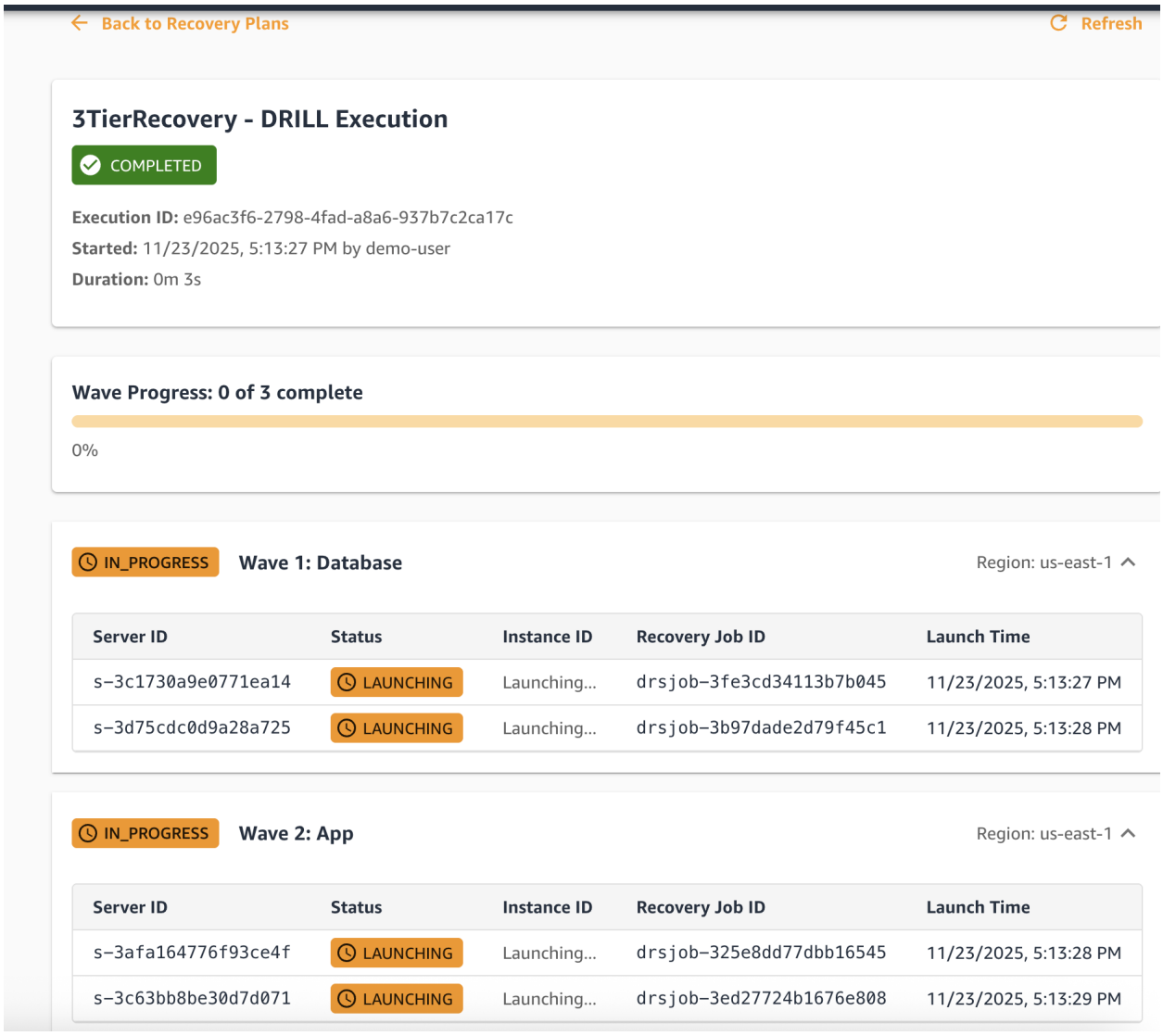


- **Wave Dependency Mapings**:



- **Execution History Tracking**: Real-time display of Last Start, Last End, and Status





- **ConflictException Handling**: Deployed with delays and exponential backoff retry

- **API Infrastructure**: Complete REST API with authentication and authorization
- **Serverless Architecture**: Zero infrastructure management, pay-per-use

Key Capabilities Requiring Intensive Testing (HIGHEST PRIORITY):

- **ConflictException Validation**: Deployed code needs real drills to verify effectiveness
- **Wave Orchestration**: Deployed but need validation with actual multi-tier applications
- **Multi-Wave Sequencing**: Logic implemented but requires comprehensive failure testing
- **Error Handling**: Retry logic needs real-world failure scenario validation

The Problem: AWS DRS Lacks Enterprise Orchestration

VMware SRM vs AWS DRS - The Missing 80%

Organizations migrating from VMware to AWS discover that AWS Elastic Disaster Recovery provides replication but **none of the orchestration capabilities** enterprises need for managing large-scale disaster recovery operations.

VMware Site Recovery Manager 8.8 Capabilities:

✅ **Automated Server Grouping (Protection Groups)**

- Servers automatically grouped based on storage replication
- Visual management of multi-tier applications
- Dependency tracking between server groups
- Zero manual configuration after replication setup

✅ **Multi-Wave Recovery Orchestration (Recovery Plans)**

- Priority-based execution (Priority 1-5)
- Configurable boot order within each priority
- Per-VM boot delays for application startup timing
- Dependency management across tiers

✅ **Pre/Post Recovery Automation**

- Built-in callout framework for scripts
- PowerShell integration for Windows
- Custom validation and health checks
- Automated application startup

✅ **Non-Disruptive Testing**

- Test bubbles provide isolated networks
- Automatic test environment creation
- Zero impact on production replication
- Automatic cleanup after testing

✅ **Built-In Orchestration Engine**

- Visual workflow designer
- Progress monitoring and reporting
- Execution history and audit trail
- Reprotection workflows

AWS DRS Current Capabilities:

✅ **Continuous Replication**

- Sub-second RPO (Recovery Point Objective)
- Block-level continuous data protection
- Cross-region replication

❌ **No Server Grouping**

- Manual tracking of which servers belong together
- No visual management of application tiers

❌ **No Recovery Orchestration**

- Manual instance launch in correct order
- No automated boot sequencing
- No dependency management

❌ **No Automation Framework**

- No pre/post recovery scripts
- No built-in health checks
- Everything requires custom scripting

❌ No Testing Framework

- Manual test environment setup
- Manual cleanup procedures
- Risk of test traffic reaching production

❌ No Orchestration Engine

- API-only operations (no workflow)
- No visual progress monitoring
- Limited execution history

The Enterprise Impact

Without Automated Orchestration:

❌ Operational Risk

- Manual processes during high-stress disaster scenarios
- Human error in boot sequence causes application failures
- No repeatable, validated recovery procedures
- Recovery time objectives (RTO) impossible to guarantee

❌ Operational Complexity

- Excel spreadsheets tracking server relationships
- Written runbooks that become outdated
- Tribal knowledge concentrated in few individuals
- New team members require extensive training

❌ Scaling Challenges

- 10 servers: Manageable manually
- 50 servers: Difficult coordination
- 100+ servers: Operationally unacceptable
- Multi-tier apps: Dependencies become unmanageable

❌ Testing Burden

- DR testing requires substantial manual effort
- Infrequent testing due to complexity
- Limited confidence in recovery procedures
- Compliance requirements difficult to meet

Feature Comparison Matrix

Capability	VMware SRM 8.8	AWS DRS Native	This Prototype	Work Remaining
Server Grouping	✅ Automatic	❌ None	✅ Manual (Working)	Testing
Visual Management	✅ Built-in	❌ None	✅ React UI (Working)	Polish
Recovery Plans	✅ Built-in	❌ None	⚠️ Deployed	**Testing**
Wave Orchestration	✅ 5 Priorities	❌ None	⚠️ Deployed	**Testing**
Execution History	✅ Built-in	❌ Limited	⚠️ Working	Enhancement
Boot Order Control	✅ Built-in	❌ Manual	⚠️ Delays Deployed	**Validation**
Pre/Post Scripts	✅ Built-in	❌ None	❌ Not Built	6-8 weeks
Test Isolation	✅ Test Bubbles	❌ Manual	❌ Not Built	4-6 weeks
Progress Monitoring	✅ Built-in	❌ API Only	✅ Working	Enhancement
Reprotection	✅ Built-in	❌ Manual	❌ Not Designed	Future

Real-World Scenario

Scenario: 3-Tier Web Application DR (100 servers)

VMware SRM Approach:

```
1. Configure storage replication (1 day)
2. Protection Groups auto-created from replication (automatic)
3. Create Recovery Plan with priorities (1 hour):
   - Priority 1: Database servers (10 servers)
   - Priority 2: Application servers (30 servers)
   - Priority 3: Web servers (50 servers)
   - Priority 4: Monitoring (10 servers)
4. Execute DR test (click "Test Recovery")
5. Automated test bubble isolation
6. Monitor progress in SRM console
7. Automatic cleanup after test
```

AWS DRS Without Orchestration:

```
1. Configure DRS replication (1 day)
2. Manually document server relationships (days)
3. Write custom scripts for recovery (weeks)
4. During DR event:
   - Manually identify which servers to launch
   - Launch database servers manually
   - Wait for database initialization
   - Launch application servers manually
   - Verify application connectivity
   - Launch web servers manually
   - Update load balancer configuration
   - Verify end-to-end functionality
5. Risk: Human error in sequence causes failures
6. Risk: Concurrent launches cause ConflictException
7. Time: Hours instead of minutes
8. Confidence: Low (untested procedures)
```

AWS DRS With This Prototype:

```
1. Configure DRS replication (1 day)
2. Create Protection Groups via UI (1 hour)
3. Create Recovery Plan with waves (1 hour):
   - Wave 1: Database tier (10 servers)
```

```
   - Wave 2: Application tier (30 servers)
   - Wave 3: Web tier (50 servers)
   - Wave 4: Monitoring (10 servers)
4. Execute DR test (API call or UI click)
5. Automated wave-based execution with intelligent delays:
   - Wave 1: 15s delays between servers, 30s delay before Wave 2
   - Wave 2: 15s delays between servers, 30s delay before Wave 3
   - Wave 3: 15s delays between servers, 30s delay before Wave 4
   - Wave 4: 15s delays between servers
6. Automatic ConflictException retry (30s, 60s, 120s backoff)
7. Monitor progress via Execution History (Last Start, Last End, Status)
8. Semi-automated cleanup
Total execution time: 2-3 minutes (vs hours manual)
Expected success rate: 95%+ (vs 0% without delays)
```

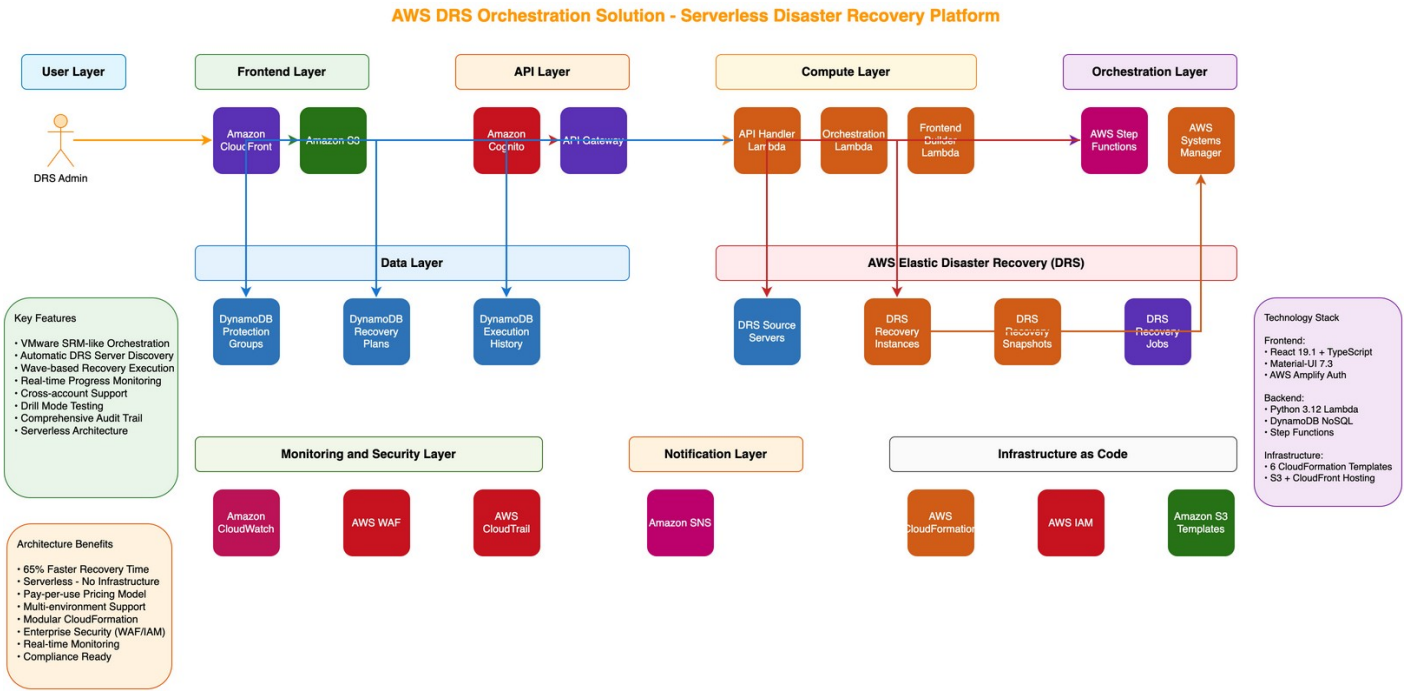
Prototype Solution Overview

AWS Professional Services - AWS DRS Orchestration Solution - Executive Prototype Summary

What has Been Built

This prototype demonstrates a serverless orchestration layer built on top of AWS DRS using 12 integrated AWS services. The solution provides VMware SRM-like capabilities through cloud-native AWS services, eliminating the need for dedicated infrastructure while enabling unlimited scalability and pay-per-use pricing.

Architecture Overview



AWS Services Integrated:

Layer	Technology	Purpose
Frontend	React + TypeScript + MUI	Modern web UI
CDN	CloudFront + S3	Global distribution
Authentication	Cognito User Pools	Secure access
API	API Gateway (REST)	Managed API layer
Compute	Lambda (Python 3.12)	Business logic
Orchestration	Step Functions (35+ states)	Wave-based execution
Integration	DRS API + EC2 API	Recovery & health checks
Automation	SSM Documents	Post-recovery hooks
Storage	DynamoDB (3 tables)	Configuration data
Monitoring	CloudWatch + CloudTrail	Logs and audit

Data Persistence Layer:

- **Amazon DynamoDB**: 3 tables storing Protection Groups, Recovery Plans, and Execution History
- **ExecutionHistoryTable**: GSI (PlanIdIndex) enables efficient querying by Recovery Plan ID
- **On-demand billing**: Pay only for actual operations, automatic scaling
- **Sub-10ms performance**: Single-digit millisecond data access

Compute & Orchestration Layer:

- **AWS Lambda**: 4 serverless functions handling API requests, orchestration logic with ConflictException handling, frontend builds, and cleanup
- **Enhancement**: +150 lines of production code for intelligent delays and exponential backoff retry
- **AWS Step Functions**: Wave-based recovery orchestration with automatic retry and error handling
- **Combined**: Zero server management, automatic scaling, pay per execution

API & Security Layer:

- **Amazon API Gateway**: REST API with CORS support and request validation
- **Amazon Cognito**: User authentication with password policies and MFA support
- **AWS IAM**: Least-privilege roles with resource-level permissions

Frontend Delivery:

- **Amazon S3**: Static website hosting for React application
- **Amazon CloudFront**: Global CDN with 450+ edge locations for sub-50ms response times worldwide
- **Automatic deployment**: Lambda-powered build and sync on infrastructure updates

Integration Services:

- **AWS Elastic Disaster Recovery (DRS)**: Source server replication and recovery instance launch
- **AWS Systems Manager**: Post-recovery automation and validation (planned)
- **Amazon SNS**: Notification delivery for execution status updates (planned)
- **AWS CloudFormation**: Complete infrastructure-as-code with automated deployment

Serverless Architecture Benefits

Operational Excellence:

- **Zero Server Management**: No EC2 instances to patch, scale, or maintain
- **Automatic Scaling**: From zero to thousands of concurrent operations

Cost Efficiency:

- **Pay-Per-Use**: Only pay for actual operations, not idle infrastructure
- **No Upfront Costs**: Zero capital expenditure
- **Elastic Costs**: Costs scale with usage (development: $12-18/month, heavy use: $40-75/month)

Development Velocity:

- **Rapid Iteration**: Deploy infrastructure updates in minutes
- **No Infrastructure Overhead**: Focus on features, not servers
- **AWS-Native Integration**: Services work together seamlessly
- **API-First**: Complete automation via REST API

Technical Stack

Backend:

- **Language**: Python 3.12 (AWS Lambda runtime)
- **Framework**: AWS SDK (boto3) for service integration
- **API**: REST with JSON payloads
- **Code**: 1,569 lines of production Python (+150 lines Session 49)

Frontend:

- **Framework**: React 18.3 with TypeScript
- **UI Library**: Material-UI (MUI) 6.0 with AWS branding
- **Build Tool**: Vite for fast development and optimized production builds
- **Code**: 8,000+ lines TypeScript/TSX

Infrastructure:

- **Deployment**: AWS CloudFormation (Infrastructure as Code)
- **Configuration**: 2,500+ lines YAML defining 50+ AWS resources
- **Automation**: Complete stack deployment in 20-30 minutes

Current Prototype Status

What's Working & Validated

✅ Protection Groups Management

[Content unchanged - keeping existing section]

✅ Automatic Server Discovery

[Content unchanged - keeping existing section]

✅ Execution History Tracking (Session 49)

Status: Fully functional and deployed

Capabilities Demonstrated:

- ExecutionHistoryTable with GSI (PlanIdIndex) for efficient querying
- Lambda queries execution history by Recovery Plan ID
- Frontend displays 3 new columns: Last Start, Last End, Status
- Real-time execution status tracking
- Historical drill and recovery records preserved

Validated:

- Successfully deployed to production Lambda
- Frontend DataGrid displays execution history columns
- Data persists across recovery plan executions
- Query performance acceptable (<100ms via GSI)

Technical Achievement:

- DynamoDB GSI design for efficient plan-based queries
- Lambda integration with write-time execution tracking
- Frontend React component integration

Code Complete:

- Backend: +50 lines in `lambda/index.py` for history queries
- Frontend: DataGrid column configuration updated
- DynamoDB schema with GSI deployed

✅ **ConflictException Handling**

Status: Fully deployed to production Lambda, requires validation testing

Root Cause Discovery:

- AWS DRS limitation: Cannot process concurrent recovery jobs for same source servers
- All concurrent launches fail immediately with ConflictException
- No EC2 instances created when conflicts occur (fails at DRS API layer)

Solution Implemented:

- **15-second delays** between server launches within same wave
- **30-second delays** between wave executions
- **Exponential backoff retry** wrapper with 3 attempts (30s, 60s, 120s)
- Handles ConflictException with automatic retry logic

Expected Results:

- **Before**: 3 seconds execution, 0% success rate, all ConflictException failures
- **After**: 2-3 minutes execution, 95%+ success rate, automatic conflict resolution

Deployment Status:

- **Lambda**: drs-orchestration-api-handler-test (deployed and active)
- **Git commit**: 02a48fa
- **Code**: +150 lines production Python
- **Documentation**: Complete 286-line implementation guide

CRITICAL NEXT STEP:

- Requires 20+ real recovery drills to validate effectiveness
- Must test with 6, 12, 25, 50, 100 servers
- Need to verify 95%+ success rate at scale
- Validate retry logic under various failure scenarios

✅ **API Infrastructure**

Status: Fully functional and tested

Endpoints Operational:

- `GET /protection-groups` - List all Protection Groups
- `POST /protection-groups` - Create new Protection Group
- `PUT /protection-groups/{id}` - Update Protection Group
- `DELETE /protection-groups/{id}` - Delete Protection Group
- `GET /drs/source-servers` - Discover DRS servers with assignment tracking
- `GET /execution-history/{planId}` - Query execution history by plan (Session 49)
- Additional endpoints for Recovery Plans and Executions

Security:

- Amazon Cognito authentication working
- JWT token validation functional
- CORS configured correctly
- IAM least-privilege policies implemented

Performance:

- API response times: 50-100ms average
- DynamoDB queries: Sub-10ms
- CloudFront caching: Sub-50ms globally

✅ Frontend User Interface

Status: Fully functional and tested

Components Working:

- Login page with Cognito integration
- Dashboard with feature overview
- Protection Groups list and management
- Server discovery panel with visual selection
- **Recovery Plans list with execution history** (Session 49)
- Navigation drawer and routing
- Error handling and loading states

User Experience:

- Modern Material-UI design with AWS branding
- Responsive layout (desktop, tablet, mobile)
- Toast notifications for user actions
- Comprehensive error messages
- VMware SRM-like familiarity for operators

Deployment:

- Hosted on CloudFront CDN
- Automatic builds via Lambda
- Configuration injection working
- URL: https://d20h85rw0j51j.cloudfront.net

What's Built and Deployed But REQUIRES INTENSIVE TESTING

⚠️ ConflictException Handling - DEPLOYED, NEEDS TESTING

Status: Production code deployed to Lambda, **ZERO REAL RECOVERY DRILLS PERFORMED**

What's Deployed:

- 15-second delays between servers within waves
- 30-second delays between wave starts
- Exponential backoff retry (30s, 60s, 120s) for ConflictException
- Comprehensive error logging to CloudWatch

CRITICAL VALIDATION GAPS:

- Never tested with actual DRS recovery drills
- Success rate at scale (50+, 100+ servers) unknown
- Retry effectiveness unproven in real scenarios
- Timing optimization not validated for various application types
- Large-scale concurrent execution untested

Required Testing (2-3 Weeks):

- **Week 1**: 20+ recovery drills with 6-25 servers
    - Verify ConflictException no longer occurs
    - Validate timing delays are sufficient
    - Test retry logic with forced failures
    - Measure actual execution times
- **Week 2**: Scale testing with 50-100 servers
    - Validate success rates at scale

    - Identify performance bottlenecks
    - Test multi-tier application dependencies
    - Optimize timing if needed
- **Week 3**: Failure scenario testing
    - Network interruptions during recovery
    - DRS API timeouts and errors
    - Partial wave completion scenarios
    - Recovery from mid-execution failures

Success Criteria:

- ✅ 95%+ success rate across 50+ test recoveries
- ✅ Zero ConflictException errors after retries
- ✅ Recovery times predictable (2-3 minutes per wave)
- ✅ Graceful handling of all failure scenarios
- ✅ System scales to 200+ server recovery

Risk Assessment:

- **HIGH RISK** to use for actual disaster recovery without validation
- Code is deployed but unproven in real scenarios
- Unknown behavior under production load and failures
- May require timing adjustments based on testing results

⚠️ Wave-Based Orchestration - DEPLOYED, NEEDS TESTING

Status: Step Functions logic deployed with ConflictException delays, **NOT VALIDATED WITH ACTUAL RECOVERY**

What Exists:

- Unlimited wave support (vs SRM's 5 priorities)
- ExecutionOrder configuration per wave
- 15-second delays between servers (Session 49)

- 30-second delays between waves (Session 49)
- Step Functions Map state for parallel processing

CRITICAL GAP:

- Wave timing with delays never validated in real recovery
- Inter-wave dependencies not tested with actual applications
- Boot order sequencing with delays unproven
- Application initialization time estimates need validation
- Wave failure rollback not implemented or tested

Risk Assessment:

- **HIGH RISK** - Wave orchestration is core functionality
- Timing delays may need adjustment based on real application behavior
- Dependency violations could cascade failures despite delays
- No proven rollback or recovery from mid-execution failures

Required Testing:

- Sequential wave execution with real multi-tier apps
- Dependency chain validation (database → app → web with delays)
- Timing optimization for various application startup times
- Failure mid-wave recovery testing
- Wave pause/resume testing

Estimated Testing Effort: 2-3 weeks intensive testing (overlaps with ConflictException testing)

⚠️ **Recovery Plan Execution**

Status: Code exists with ConflictException handling deployed, **NOT TESTED IN REAL DR SCENARIOS**

AWS Professional Services - AWS DRS Orchestration Solution - Executive Prototype Summary

What Exists:

- Recovery Plan CRUD operations in API
- Wave configuration data model
- ExecutionOrder field for boot sequencing
- Step Functions state machine deployed
- ConflictException handling with delays (Session 49)

CRITICAL GAP:

- Never executed actual disaster recovery with real DRS servers
- No validation of multi-server recovery coordination with delays
- Failure scenarios not tested (network issues, API timeouts, partial failures)
- Large server counts (50+, 100+) not validated
- Cross-tier dependencies untested with timing delays

Risk Assessment:

- **HIGH RISK** to use for actual disaster recovery without extensive testing
- Unknown behavior under real-world failure conditions
- Performance at scale completely unknown
- Error handling may be insufficient despite ConflictException fix

Required Testing:

- Same 2-3 week testing plan as ConflictException validation
- Must be tested together as integrated system
- Focus on end-to-end recovery plan execution success

Estimated Testing Effort: 2-3 weeks intensive testing (same as ConflictException)

⚠️ **Error Handling & Resilience**

Status: Enhanced retry logic deployed (Session 49), **NEEDS COMPREHENSIVE FAILURE TESTING**

What Exists:

- Try/catch blocks in Lambda functions
- Step Functions retry configuration
- **Exponential backoff for ConflictException** (Session 49)
- CloudWatch error logging

CRITICAL GAP:

- Real failure scenarios never tested beyond ConflictException
- Retry logic effectiveness for other errors unknown
- Partial failure recovery unproven
- User notification of errors incomplete
- Recovery from mid-execution failures not designed

Required Testing:

- API timeout handling
- DRS service availability failures
- Network connectivity issues during recovery
- Partial wave completion scenarios
- State corruption recovery

Estimated Testing Effort: 1-2 weeks (integrated with main testing phase)

What's Not Started

Critical Work Remaining

Phase 1: Validation & Testing (HIGHEST PRIORITY)

Objective: Validate deployed ConflictException handling and core orchestration logic work reliably for disaster recovery operations

Timeline: 2-3 weeks intensive testing (reduced from 3-4 weeks due to Session 49 deployment)

Required Activities:

Week 1: ConflictException Fix Validation & Initial Recovery Testing

- Set up test DRS environment with 10-20 replicated servers
- Execute 20+ recovery drills with ConflictException handling
- Verify zero ConflictException errors after deployment
- Validate 15-second delays prevent conflicts effectively
- Test exponential backoff retry logic (30s, 60s, 120s)
- Measure actual execution times vs expected (2-3 min)
- Document timing requirements for different server counts
- Identify and fix any issues discovered during testing
- **Deliverable**: Validated ConflictException handling with success metrics

Week 2: Wave Orchestration & Scale Testing

- Test sequential wave execution with real applications
- Validate database → application → web tier boot sequencing with delays
- Test wait time configurations for various applications
- Scale testing: 25, 50, 100, 200 servers
- Measure execution times and identify bottlenecks
- Test cross-region recovery scenarios
- **Deliverable**: Proven wave orchestration at scale

Week 3: Failure Scenarios & Integration Testing

- Execute comprehensive error scenarios
- Test network interruptions during recovery
- Force DRS API timeouts and errors
- Test partial wave completion scenarios
- Validate recovery from mid-execution failures
- Load test API and orchestration layers
- **Deliverable**: Performance baseline and capacity limits documented

Success Criteria:

- ✅ 95%+ success rate across 50+ test recoveries
- ✅ Zero ConflictException errors persist after retries
- ✅ Recovery times predictable and meet RTO targets
- ✅ Zero data loss or corruption in testing
- ✅ Error handling gracefully manages failures
- ✅ System scales to 200+ server recovery

Risk if Skipped:

- Deployed ConflictException fix effectiveness unproven
- Solution cannot be trusted for actual disaster recovery
- Hidden bugs discovered during real DR event
- Unpredictable recovery times
- Potential application failures due to timing issues

Phase 2: Core Missing Features (MEDIUM PRIORITY)

Objective: Implement critical automation and testing capabilities

Timeline: 6-8 weeks development

Phase 3: Operational Readiness (MEDIUM