TERMINOLOGY

Thursday, September 11, 2025 4:43 PM

ACRONYMS

Thursday, September 11, 2025 4:44 PM

PDD	Policy Detail Database	
QAS	Quote & Application System	
IRPM	Individual Risk Premium Modification	
CIM	Customer Information Management	
MDM	Master Data Management	
STG BILLING	Strategic Technology Group	
EDH	Erie Data Hub	
FOOD	File Of Our Dream	
RS3	Reporting System 3	
DNB	Dun & Bradstreet	
ECC	Erie Claims Center	
OOB	Out-of-Balance	
OOS	Out-of-Sequence	
OOSE	Out-of-Sequence Endorsement	OOSE is a policy change or endorsement that is dated earlier than a current active transaction. It occurs out of chronological order, requiring the system to revert and reapply subsequent transactions to maintain consistency.
TRF	Term Roll Forward	TRF is a process that extends a policy term into a new period, typically at renewal. It involves carrying forward balances, coverages, and endorsements from the current term to the next.
BPP	Business Personal Property	
OSPAS		
TESL	Transaction Engine Service Layer	TESL is a Service-Oriented Architecture layer that exposes One Shield's core function as Web Service (SOAP/REST). It allows external systems (portal/mobile apps/third-party tools) to create, read, updated, delete, and search insurance data. Key capabilities of TESL are - • Headless Processing: Decoupling business logic from the UI, allowing backend services to be triggered independently. • API-Driven Architecture: Offers a rich set of prebuilt APIs for Policy, Billing, Claims, and Customer Data. • Metadata Services: Exposes product definitions (Coverages, Rules, Forms, etc.) to external systems. • Portal Enablement: Powers self-service portals for Agents, Customers, and Partners. • Rapid Integration: Simplified and accelerates integration with CRM's ERP's, and other insurance.
SOA	Service-Oriented Architecture	
FCRA	Fair Credit Reporting Act	
CLUE	Comprehensive Loss Underwriting Exchange	It shows the prior Auto and Property claims history.
MVR	Motor Vehicle Records	
DMV	Departments of Motor Vehicle	
ESB	Erie Secure Business	
QAS	Quote & Apps System	
MPP	Miss Process Policy	It is about how to handle the data after the fix for the records impacted prior to the fix - Calendar Date vs Effective Date
BOX	Business Owner Experience	Non-Regulatory Changes
BOP	Business Owner Products	Regulatory Changes

SERVICES

Thursday, September 11, 2025 4:45 PM

Questions

Thursday, September 11, 2025 4:58 PM

Calendar Effective Date vs Policy Effective Date?

Lexius Nexis Service ?

ISO Bureau Content ?

ACORD Messaging System and London Exchange for RI ?

CDMP

Friday, September 12, 2025 9:43 AM

- **Data Governance**
- **Data Quality**
- **Data Modeling & Design**
- **Metadata Management**
- **Master & Reference Data Management**
- **Data Integration & Interoperability**
- **Data Warehousing & Business Intelligence**

From <https://www.datastrategypros.com/resources/cdmp-specialist-exam>

NOTES

Friday, September 12, 2025 2:42 PM

Property Coverage	Protects the policyholder's own property - such as their home, car, or personal belongings.
Liability Coverage	Protects the policyholder from claims involving others - specifically for bodily injury or property damage they cause to someone else.
Schedule Rating	Schedule Rating is an insurance tool that allows insurers to adjust premium based on specific characteristics of a business, beyond what's reflected in standard rates or past claims history. It involves applying credits or debits to a base premium to account for factors like safety measures, management practices, and the physical condition of the business.
Observability	Observability in software development in the ability to understand a system's internal state by analyzing its external output. It's about gaining insight into how a system is performing, identifying issues, and ensuring smooth operations, especially in complex, distributed systems. Observability relies on collecting and analyzing telemetry data like logs, metrics, and traces, allowing teams to understand the system's behavior and diagnose problems effectively. Here's a more detailed breakdown - • Understanding Internal State: Observability goes beyond simply monitoring for predefined alerts. It allows teams to delve into the system's inner workings by examining its outputs. • Telemetry Data: Key to observability are logs, metrics, and traces. Logs provide detailed records of events, metrics offer quantitative data, and traces map the flow of requests across different components. • Beyond Traditional Monitoring: Observability provides a more flexible and comprehensive approach that traditional monitoring, which often relies on pre-defines metrics and thresholds. • Troubleshooting and Debugging: Observability tools and techniques help teams quickly identify the root cause of issues in distributed systems, enabling faster resolution and preventing future incidents. • Continuous Improvement: By providing deep insights into system behavior, observability empowers teams to optimize performance, enhance reliability, and facilitate continuous delivery.

ERIE

Friday, September 12, 2025 2:51 PM

ESB (ErieSecure Business) is a new single commercial package policy combining - Ultrapack Plus, Ultraflex, Fivestar, Commercial Fire, General Liability, Inland Marine, Crime.

- **Ultrapack Plus:** Ultrapack Plus is a business insurance policy offered by Erie Insurance designed for small to medium-sized business. It's a customizable package policy that can be tailored to various industries, including retail, wholesale, office, and service businesses. Ultrapack Plus includes coverage for property, liability, and can be enhanced with optional endorsements like equipment breakdown coverage and food contamination protection.
- **Fivestar:** This policy is bundled insurance solution for contractors, simplifying the process of getting coverage for different risks. It includes property coverage for building and contents, general liability protection, inland marine coverage for equipment, and building risk coverage. It can be customized with additional coverage options like tools, contractors' equipment, and more.
- **Commercial Fire:** Commercial Fire insurance, often included within a commercial property insurance policy or a Business Owner's Policy (BOP), safeguards the business's assets against fire and smoke damage.
- **General Liability:** General Liability insurance, also known as commercial general liability (CGL) or business liability insurance, is a fundamental type of insurance for businesses of all sizes. It provides financial protection against various common risks and lawsuits arising from day-to-day business operations. It covers bodily injury, personal & advertising injury, and property damage.
- **Inland Marine:** Inland Marine insurance, sometimes called a "Floater Policy" is a type of business insurance that covers goods, materials, tools, and equipment that are transported over land. It can also protect property that is temporarily stored at off-site locations or owned by others but under the care of your business. This type of policy evolved from ocean marine insurance, which covers items transported by sea. Inland marine insurance fills the gaps in coverage for products, materials, or equipment as they are transported on land, such as by truck or train.
- **Crime:** Crime insurance, also known as commercial crime insurance, protects business from financial losses due to various criminal acts committed by both employees and third parties. These act can range from theft and fraud to embezzlement and forgery. It covers Employee Theft/Dishonesty, Forgery/Alterations, Computer Fraud, Fund Transfer Fraud, Theft of Money and Securities, Robbery & Burglary, Social Engineering Fraud, Kidnapping/Ransom, Extortion.

2. Insurance

Monday, September 15, 2025 9:31 AM

PERSONAL

- Private Passenger Auto
- ErieSecure Home
- Personal Catastrophe Liability
- Boat Protector
- Mobile HomeProtector
- Dwelling Property/Liability
- Ultrasure for Landlords
- Personal Umbrella

COMMERCIAL

- Commercial Umbrella
- Commercial Auto
- CMP & Other (Ino new business)
- ErieSecure Business
- Workers Compensation
- Bonds

LIFE

- ERIExpress Life
- Whole/Universal/Term/Group
- Annuities/IRA
- Medicare Supplement

Knowledge of ISO Commercial Lines Product Offerings, Rating and Forms

Experience in Loss Sensitive insurance programs with WorkComp, Commercial Auto, and General Liability including retrospective rating plans, large deductible plans, and self-insured retentions.

2a. P&C Insurance

Friday, September 19, 2025 8:54 AM

2b. Health Insurance

Friday, September 19, 2025 8:54 AM

2c. Life Insurance

Friday, September 19, 2025 8:54 AM

2d. Reinsurance

Friday, September 19, 2025 8:55 AM

2e. Compliance

Friday, September 19, 2025 8:53 AM

HIPPA
GDPR

1. SYSTEM DESIGN

Friday, September 12, 2025 3:56 PM

NGNIX	It is a high-performance, open-source web server software that also functions as a Reverse Proxy, Load Balancer, HTTP Cache, Mail Proxy. It is popular because of its high performance with low resource usage, event-driven architecture (asynchronous), scalability, and easy configuration with strong community support. • Web Server: Serves static content like HTML, CSS, JS, and images. • Reverse Proxy: Forwards client requests to backend servers (e.g., Node.js, Python, Java apps). • Load Balancer: Distributes traffic across multiple servers to improve performance and reliability. • API Gateway: Manages and routes API requests. • SSL Termination: Handles HTTPS encryption/decryption before passing requests to backend servers. In a modern web application stack, Nginx might sit in front of application servers (like Flask, Express, or Spring Boot), handling incoming traffic, serving static files, and forwarding dynamic requests.

Module 1: Architecture & Design Patterns Microservices vs Monolith, Domain-Driven Design (DDD), CQRS & Event Sourcing, 12-Factor App Principles, Design for Failure/Resilience	• Software Architecture Styles: Monolith, Microservices, Serverless, Layered, Hexagonal, Event-Driven • Design Patterns: Singleton, Factory, Adapter, Observer • Enterprise Patterns: CQRS, Event Sourcing, Saga • Domain-Driven Design (DDD), 12-Factor App Principles • Design for Failure, Resilience, Anti-Patterns • SOLID, DRY, KISS, YAGNI principles	Duration: 4–6 weeks Goals: Understand foundational architecture styles and design patterns for scalable, maintainable systems. • Software Architecture Styles: Monolith, Microservices, Serverless • Design Patterns: Singleton, Factory, Adapter, Observer • Enterprise Patterns: CQRS, Event Sourcing, Saga • Domain-Driven Design (DDD) • 12-Factor App Principles • Design for Failure / Resilience • Microservices vs Monolith
Module 2: API Architecture & Integration Message Queue (Kafka, RabbitMQ), Event-Driven Architecture, Service Mesh (Istio, Linkerd), API Gateway (Kong, Apigee, Azure API Management)	• API Fundamentals ○ REST, SOAP, GraphQL, gRPC, WebSockets, Webhooks ○ Stateless vs Stateful APIs ○ Synchronous vs Asynchronous APIs • API Design & Documentation ○ OpenAPI/Swagger ○ Versioning, Pagination, Rate Limiting • API Management ○ API Gateway (e.g., Apigee, Azure API Management) ○ Throttling, Caching, Monitoring • Integration Patterns ○ Event-Driven Architecture ○ Message Queues (Kafka, RabbitMQ) ○ Service Mesh (Istio, Linkerd) ○ Composite APIs • API Testing: Postman, RestAssured	Duration: 4–6 weeks Goals: Understand API types, design principles, and integration patterns. API Fundamentals: REST, SOAP, GraphQL, gRPC, WebSockets, Webhooks Communication Styles: Stateless vs Stateful, Synchronous vs Asynchronous API Design & Documentation: OpenAPI/Swagger, Versioning, Pagination, Rate Limiting API Management: API Gateway (Kong, Apigee, Azure API Management), Throttling, Caching, Monitoring Integration Patterns: Event-Driven Architecture, Message Queues (Kafka, RabbitMQ), Service Mesh (Istio, Linkerd), Composite APIs Resources: Google API Design Guide freeCodeCamp REST API Tutorial GraphQL Official Docs gRPC Basics – grpc.io WebSockets Tutorial – MDN
Module 3: Logging, Monitoring & Observability Observability, log aggregation, metrics, tracing (e.g., ELK, Prometheus, Grafana)	• Logging Best Practices ○ Structured Logging ○ Centralized Log Management (ELK, Fluentd) • Monitoring & Metrics ○ Prometheus, Grafana ○ Application Performance Monitoring (APM) tools ○ Tracing & Observability ○ Distributed Tracing (Jaeger, Zipkin, OpenTelemtry) ○ Health Checks & Alerting, Synthetic Monitoring, RUM ○ Log Retention & Compliance Logging	Duration: 3–4 weeks Goals: Learn centralized logging, metrics, tracing, and alerting. • Logging Best Practices: Structured Logging, Centralized Log Management (ELK, Fluentd) • Monitoring & Metrics: Prometheus, Grafana, APM tools • Tracing & Observability: Distributed Tracing (Jaeger, Zipkin), Health Checks, Alerting Resources: • Honeycomb Observability Guide • Prometheus + Grafana Tutorial – DigitalOcean • ELK Stack Guide – Elastic
Module 4: Data Architecture & Management Data modeling, data governance, ETL/ELT, data lakes, data warehouses	• Data Modeling & Storage ○ Relational vs NoSQL ○ Data Lakes, Warehouses • Data Integration ○ ETL/ELT Pipelines ○ Streaming Data (Apache Kafka, Azure Event Hubs) • Data Governance ○ Metadata Management ○ Master Data Management (MDM) ○ Data Lineage, Cataloging • Data Privacy & Compliance ○ GDPR, HIPAA, Data Residency • Advance Concepts ○ Data Mesh ○ Data Fabric ○ Data Quality	Duration: 4–6 weeks Goals: Understand data modeling, storage, ETL/ELT, governance, and compliance. • Data Modeling & Storage: Relational vs NoSQL, Data Lakes, Warehouses • Data Integration: ETL/ELT Pipelines, Streaming Data (Kafka, Azure Event Hubs) • Data Governance: Metadata Management, Master Data Management (MDM) • Data Privacy & Compliance: GDPR, HIPAA, Data Residency Resources: • Data Engineering Zoomcamp • Google Cloud Big Data & ML Fundamentals (Coursera – Free Audit) • Talend Data Governance Basics
Module 5: Security Architecture Authentication (OAuth2, SAML), Authorization (RBAC/ABAC), encryption, API security, secrets management.	• Authentication & Authorization ○ OAuth2, OpenID Connect, SAML ○ RBAC vs ABAC • API Security ○ JWT, API Keys, HMAC ○ OWASP Top 10 for APIs • Infrastructure Security ○ Network Segmentation, Firewalls ○ Secrets Management (Vault, Azure Key Vault) • Compliance & Governance ○ Audit Logging ○ Security Policies & Controls ○ Zero Trust Architecture • Security Testing: SAST, DAST, Penetration Testing	Duration: 4–5 weeks Goals: Learn authentication, authorization, API security, and compliance. • Authentication & Authorization: OAuth2, OpenID Connect, SAML, RBAC vs ABAC • API Security: JWT, API Keys, HMAC, OWASP Top 10 • Infrastructure Security: Network Segmentation, Firewalls, Secrets Management (Vault, Azure Key Vault) • Compliance & Governance: Audit Logging, Security Policies & Controls Resources: • OWASP API Security Top 10 • OAuth 2.0 Simplified – Aaron Parecki • Google Cloud Security Fundamentals (Coursera – Free Audit)
Module 6: Performance & Scalability Caching, load balancing, latency optimization, scalability, throughput.	• Performance Optimization ○ Caching Strategies (Redis, CDN) ○ Load Balancing • Scalability Patterns ○ Horizontal vs Vertical Scaling	Duration: 3–4 weeks Goals: Learn caching, load balancing, scaling strategies, and resilience. • Performance Optimization: Caching Strategies (Redis, CDN), Load Balancing • Scalability Patterns: Horizontal vs Vertical Scaling, Auto-scaling, Sharding • Resilience & Reliability: Circuit Breakers, Retry Logic, Chaos Engineering

	○ Auto-scaling, Sharding • Resilience & Reliability ○ Circuit Breakers, Retry Logic ○ Chaos Engineering • Benchmarking & Load Testing: JMeter, Gatling • Capacity Planning, Latency vs Throughput	Resources: • High Scalability Blog • System Design Primer – GitHub • Cloudflare Caching Strategies
Module 7: Cloud Architecture IaaS, PaaS, SaaS, cloud-native design, multi-cloud, hybrid cloud, containers, serverless	• Cloud Fundamentals • Cloud Models ○ IaaS, PaaS, SaaS ○ Public, Private, Hybrid, Multi-cloud • Cloud-Native Design ○ Containers (Docker), Orchestration (Kubernetes) ○ Serverless (Azure Functions, AWS Lambda) • Cloud Services ○ Storage, Compute, Networking ○ Identity & Access Management (IAM) • Governance: Policy-as-Code, Cloud Security Posture Management (CSPM) • Cost Management: FinOps, Budgeting	Duration: 6–8 weeks Goals: Understand cloud models, services, containers, serverless, and IAM. • Cloud Fundamentals: IaaS, PaaS, SaaS, Public, Private, Hybrid, Multi-cloud • Cloud-Native Design: Containers (Docker), Orchestration (Kubernetes), Serverless (Azure Functions, AWS Lambda) • Cloud Services: Storage, Compute, Networking, Identity & Access Management (IAM) Resources: • Azure Fundamentals – Microsoft Learn • AWS Cloud Practitioner Essentials – Free • Kubernetes Basics – kube.academy
Module 8: Deployment, DevOps & Automation CI/CD pipelines, blue-green deployments, canary releases, infrastructure as code (IaC) Infrastructure as Code (Terraform, ARM templates), Configuration Management (Ansible, Chef), Containerization (Docker, Kubernetes), Monitoring & Alerting Automation	• CI/CD Pipelines ○ GitHub Actions, Azure DevOps, Jenkins • Deployment Strategies ○ Blue-Green, Canary, Rolling Updates • Infrastructure as Code (IaC) ○ Terraform, ARM Templates, Bicep • Configuration Management ○ Ansible, Chef, Puppet • GitOps, Secrets in CI/CD, Release Orchestration (Spinnaker, ArgoCD)	Duration: 4–6 weeks Goals: Learn CI/CD, deployment strategies, IaC, and configuration management. • CI/CD Pipelines: GitHub Actions, Azure DevOps, Jenkins • Deployment Strategies: Blue-Green, Canary, Rolling Updates • Infrastructure as Code (IaC): Terraform, ARM Templates, Bicep • Configuration Management: Ansible, Chef, Puppet Resources: • CI/CD with GitHub Actions – freeCodeCamp • Terraform Basics – HashiCorp Learn • Ansible for Beginners – Red Hat
Module 9: Governance & Compliance Data Privacy (GDPR, HIPPA), Audit Logging, Policy Enforcement, Cost Optimization & FinOps	• IT Governance Frameworks: COBIT, TOGAF, Zachman • Compliance Standards: GDPR, HIPAA, PCI-DSS • Risk Management (NIST RMF) • Audit Automation • Policy Enforcement & Controls • Cost Optimization & FinOps	Duration: 3–4 weeks Goals: Understand IT governance frameworks, compliance standards, and cost optimization. • IT Governance Frameworks: COBIT, TOGAF • Compliance Standards: GDPR, HIPAA, PCI-DSS • Audit & Risk Management • Policy Enforcement & Controls • Cost Optimization & FinOps
Module 10: Emerging Technologies & Trends Model Deployment & APIs, MLOps, AI Governance, Responsible AI Principles	• AI/ML Integration ○ MLOps ○ Model APIs ○ Responsible AI • Edge Computing & IoT • Platform Engineering & Internal Developer Platforms (IDPs) • Quantum Computing Basics • Sustainability in Architecture	Duration: Optional / Ongoing Goals: Stay updated with evolving tech trends and innovations. • AI/ML Integration: MLOps, Model APIs, Responsible AI • Edge Computing & IoT • FinOps & Cost Optimization • Sustainability in Architecture

M1. Architecture & Design Patterns

Friday, September 19, 2025 10:14 AM

Module 1: Architecture & Design Patterns Microservices vs Monolith, Domain-Driven Design (DDD), CQRS & Event Sourcing, 12-Factor App Principles, Design for Failure/Resilience	• Software Architecture Styles: Monolith, Microservices, Serverless, Layered, Hexagonal, Event-Driven • Design Patterns: Singleton, Factory, Adapter, Observer • Enterprise Patterns: CQRS, Event Sourcing, Saga • Domain-Driven Design (DDD), 12-Factor App Principles • Design for Failure, Resilience, Anti-Patterns • SOLID, DRY, KISS, YAGNI principles	Duration: 4–6 weeks Goals: Understand foundational architecture styles and design patterns for scalable, maintainable systems. • Software Architecture Styles: Monolith, Microservices, Serverless • Design Patterns: Singleton, Factory, Adapter, Observer • Enterprise Patterns: CQRS, Event Sourcing, Saga • Domain-Driven Design (DDD) • 12-Factor App Principles • Design for Failure / Resilience • Microservices vs Monolith

SOFTWARE ARCHITECTURE STYLES	
Monolith	A single, unified codebase where all components are tightly coupled. Easier to develop initially but harder to scale and maintain. It is a single-tiered software application where all components (UI, business logic, data access) are tightly integrated and run as a single service. Example - An early version of an e-commerce website where product catalog, user management, payment processing, and order tracking are all part of one codebase and deployed together. Pros: • Simple to develop and deploy initially • Easier debugging due to single codebase Cons: • Hard to scale individual components • Difficult to maintain as the codebase grows
Microservices	Breaks down applications into small, independent services that communicate via APIs. Promotes scalability and flexibility. It is an approach where the application is composed of small, independent services that communicate over APIs. Example - Amazon uses microservices for different functionalities like product search, recommendations, payments, and shipping—each running independently. Pros: • Scalability and flexibility • Independent deployment and development Cons: • Complex service orchestration • Requires robust DevOps and monitoring
Serverless	Executes code in response to events without managing servers. Ideal for event-driven and lightweight applications. Applications are built using cloud functions that run in response to events, without managing servers. Example - A photo-sharing app where image uploads trigger a serverless function to resize and store images in cloud storage (e.g., AWS Lambda + S3). Pros: • No server management • Cost-effective for sporadic workloads Cons: • Cold start latency • Limited execution time and control
Layered (N-tier)	Organizes code into layers (e.g., presentation, business, data). It promotes separation of concerns. Example - A banking application with: UI layer for customer interaction, business layer for transaction rules, and data layer for database operations. Pros: • Separation of concerns • Easier testing and maintenance Cons: • Can become rigid and hard to scale
Hexagonal (Ports & Adapters)	Focuses on isolating the core logic from external systems using ports and adapters. Enhances testability and flexibility. It separates the core logic from external systems using ports (interfaces) and adapters (implementations). Example - A payment processing system where the core logic is isolated and can interact with different payment gateways (PayPal, Stripe) via adapters. Pros: • High testability • Easy to swap external dependencies Cons: • Requires careful design upfront
Event-Driven	Components communicate via events. Promotes loose coupling and asynchronous processing. The components communicate by producing and consuming events, often asynchronously. Example - A ride-sharing app where a ride request triggers an event, matching service listens and assigns a driver, and the notification service sends updates to the user Pros: • Loose coupling • Scalable and responsive Cons: • Complex debugging and tracing • Requires event management infrastructure

DESIGN PATTERNS	
Singleton	Ensures a class has only one instance and provides a global point of access.
Factory	Creates objects without specifying the exact class. Promotes loose coupling.

Adapter	Converts one interface into another expected by the client. Useful for legacy integration.
Observer	Allows objects to subscribe and react to events or changes in another object. Common in UI and event systems.

ENTERPRISE PATTERNS	
CQRS (Command Query Responsibility Segregation)	It separates READ and WRITE operations for better scalability and performance.
Event Sourcing	It stores state changes as a sequence of events rather than current state. It enables auditability and replay.
SAGA	It manages distributed transactions across microservices using a sequence of local transactions and compensating actions.

SOFTWARE PRINCIPLES	
SOLID	Five principles for Object-Oriented Design: • Single Responsibility • Open/Closed • Liskov Substitution • Interface Segregation • Dependency Inversion
DRY (Don't Repeat Yourself)	Avoid code duplication to improve maintainability.
KISS (Keep It Simple, Stupid)	Simplicity leads to better design and fewer bugs.
YAGNI (You Aren't Gonna Need It)	Don't implement features until they're actually needed.

OTHER CONCEPTS	
Domain-Driven Design (DDD)	• Focuses on modeling software based on the business domain. • Uses concepts like entities, value objects, aggregates, and bounded contexts.
12-Factor App Principles	• A methodology for building scalable, maintainable cloud-native apps. • Covers codebase, dependencies, config, backing services, build/release/run, processes, port binding, concurrency, disposability, dev/prod parity, logs, and admin processes.
Design for Failure & Resilience	• Design for Failure: Anticipate and gracefully handle failures (e.g., retries, fallbacks). • Resilience: Build systems that recover quickly from failures (e.g., circuit breakers, bulkheads).
Anti-Patterns	• Common but counterproductive practices (e.g., God Object, Spaghetti Code, Magic Numbers). • Recognizing and avoiding them improves maintainability and scalability.

M2. API Concepts

Wednesday, September 17, 2025 4:10 PM

Core Technical Concepts

1. API Protocols & Standards
   - REST (Representational State Transfer)
   - SOAP (Simple Object Access Protocol)
   - GraphQL
   - gRPC
   - WebSockets
2. HTTP Fundamentals
   - Methods (GET, POST, PUT, DELETE, PATCH)
   - Status Codes
   - Headers, Cookies, and Caching
   - Authentication & Authorization (OAuth2, JWT, API Keys)
3. Data Formats
   - JSON
   - XML
   - YAML
   - Protocol Buffers (for gRPC)
4. API Design Best Practices
   - Resource naming conventions
   - Versioning strategies
   - Pagination
   - Error handling
   - Rate limiting and throttling

Architecture & Design

1. API Gateway Concepts
   - Kong, Apigee, AWS API Gateway, Azure API Management
   - Routing, transformation, security, analytics
2. Microservices Architecture
   - Service decomposition
   - Inter-service communication
   - Service discovery
   - Circuit breakers and retries
3. Security
   - TLS/SSL
   - OAuth2 flows
   - OpenID Connect
   - CORS policies
   - Threat modeling (e.g., OWASP API Security Top 10)
4. Documentation & Testing
   - Swagger/OpenAPI
   - Postman
   - Insomnia
   - Contract testing (e.g., Pact)

Development & Tooling

1. Languages & Frameworks
   - Node.js (Express, NestJS)
   - Python (FastAPI, Flask)
   - Java (Spring Boot)
   - .NET (ASP.NET Core)
   - Go (Gin, Echo)
2. CI/CD Integration
   - GitHub Actions, Jenkins, Azure DevOps
   - Automated testing and deployment pipelines
3. Monitoring & Logging
   - Prometheus, Grafana
   - ELK Stack (Elasticsearch, Logstash, Kibana)
   - Distributed tracing (Jaeger, Zipkin)

Advanced Topics

1. API Lifecycle Management
   - Design → Develop → Deploy → Monitor → Retire
2. Event-Driven APIs
   - Kafka, RabbitMQ
   - Webhooks
3. Serverless APIs
   - AWS Lambda, Azure Functions
   - BFF (Backend for Frontend) pattern
4. API Monetization & Analytics
   - Usage tracking
   - Billing models
   - Developer portals

Additional Concepts

1. Domain-Driven Design (DDD)
   - Helps in designing APIs that align with business domains.
   - Useful for microservices and large enterprise systems.
2. Design Patterns for APIs
   - API Gateway Pattern
   - Backend for Frontend (BFF)
   - Adapter/Facade patterns
   - CQRS (Command Query Responsibility Segregation)
3. API Governance
   - Standardization across teams
   - Reusability and discoverability
   - Approval workflows and version control
4. DevOps & Infrastructure as Code (IaC)
   - Terraform, ARM templates, or Bicep for Azure
   - Docker & Kubernetes for containerized API deployment
   - Helm charts for API packaging
5. API Performance Optimization
   - Caching strategies (Redis, CDN)
   - Load balancing
   - Connection pooling
   - Asynchronous processing
6. Compliance & Legal Considerations
   - GDPR, HIPAA, PCI-DSS (if working with sensitive data)
   - Data residency and retention policies
7. API Product Management
   - Treating APIs as products
   - Developer experience (DX)
   - API onboarding and SDKs
8. Integration Patterns
   - Synchronous vs Asynchronous APIs
   - Event-driven vs request-response
   - API orchestration vs choreography
9. Legacy System Integration
   - Wrapping legacy systems with APIs
   - Adapters and connectors
   - ETL vs real-time APIs

1. API Protocols & Standards

-- Types of API based on Architecture/Protocol --

- REST (Representational State Transfer) - the architecture style that uses HTTP methods (GET/PUT/POST/DELETE) and are stateless and resource-based - often considered as simple and scalable.
- SOAP (Simple Object Access Protocol) - the architecture style that uses XML for messaging and are more rigid and highly secure - often considered reliable and secured.
- GraphQL - the protocol that allows the client to request exactly the data they need - it is very efficient for mobile apps and dashboards with dynamic data needs.
- gRPC (Google Remote Procedural Call) - the architecture style that uses gRPC framework to transmit the data - often used in microservices communication (especially in cloud-native environments).
  - gRPC is a high-performance, open-source RPC framework which uses Protocol Buffers (Protobuf) for serialization - it also supports multiple languages.
  - binary format (Protobuf) is faster and smaller than JSON.
  - built on HTTP/2 that supports multiplexing, streaming, and better performance.
  - supports bi-directional streaming, where client and server can send messages simultaneously.
- WebSockets - the architecture style that provides a full-duplex communication channel over a single TCP connection .
  - unlike HTTP which is a request-response based, WebSockets allows real-time two way communication (without polling)
  - often used in chat apps, live dashboards, notifications, gaming, and real-time monitoring
  - the client opens a WebSocket connection, and Server pushes updates as event occurs (no need for repeated HTTP requests).
  - it supports persistent connection, real -time data exchange, and low latency (making it ideal for event-driven applications).

-- Types of API based on Accessibility --

- Open APIs (Public APIs) - publicly available APIs that can be used without restrictions (or with minimal registration)
- Partner APIs - APIs that are shared with specific business partners and not publicly available.
- Internal APIs (Private APIs) - API's that are used within an organization to connect internal systems and services.

-- Types of API based on Use Case --

- Data APIs - the APIs that provide access to data (weather data, stock prices)
- Service APIs - the APIs that provide access to functionality (payment processing, billing service, authentication)
- Composite APIs - these API combine multiple APIs into a single call (useful in microservices architecture)

-- Types of API based on Communication Style --

- Synchronous APIs - Client waits for the response (REST)
- Asynchronous APIs - Response is sent later (Webhooks, Messaging Queues)
  - Webhooks (a.k.a Event-Driven) - pushes data to other systems based on the occurrence of an event.

1. API Protocols & Standards

-- Rate Limiting --

- Rate limiting controls how many requests a client can make on an API within a specific time window.
- Common Strategies -
  - Fixed Window:
  - Sliding Window:
  - Token Bucket:
  - Leaky Bucket:
- Rate limits can be applied based on - API key, IP address, User ID, OAuth token.
- Modern API Gateways (Azure API Management, AWS API Gateway, Kong, or Apigee) support built-in rate limiting and throttling policies.
- Rate limits can be implemented using middleware or libraries
  - In Node.js: express-rate-limit, rate-limiter-flexible
  - In Python: Flask-Limiter
  - In Java: Bucket4j, Resilience4j
- Rate limits can also be enforced using tools like NGNIX or HAProxy.
- There are tools to monitor & analyze the usage pattern to prevent abuse, identify higher-traffic clients, optimize performance (Azure Monitor, AWS CloudWatch, Prometheus, ELK Stack)

-- Throttling --

- Throttling is the enforcement mechanism that slows down or blocks requests when rate limits are exceeded.
- Common Strategies -
  - Delay or Queue Requests
  - Temporarily block the Client/IP
  - Notify the client via headers (e.g. Retry-After) - APIs often include headers to inform clients about their usage.
  - Return 429 Error Code (too many requests HTTP status)

1. API Protocols & Standards

-- HTTP Methods --

- GET - Retrieve Data
- POST - Create New Data
- PUT - Update Entire Resources
- PATCH - Update parts of Resources
- DELETE - Remove a Resource

Use GET for Reading, POST for Creating, PU/PATCH for Updating, and DELETE for Removing

-- HTTP Status Code --

- 200 - Ok | Success
- 201 - Resource Created (after a successful POST)
- 204 - No Content | Success, No Body (after DELETE or PUT)
- 400 - Bad Request (Missing Required Fields)
- 401 - Unauthorized (No Valid Authentication - missing token)
- 403 - Forbidden (Access Denied - token valid, but no permission)
- 404 - Not Found (Resource Missing)
- 500 - Internal Server Error (Server Crashed)

-- HEADER| COOKIES| CACHING --

- **Header -** Used to pass metadata with requests/response.
  - Content-Type:- tells server the format of the request body.
  - Authentication:- used for authentication
  - Accept:- tells server what format client expects.
- **Cookies -** Used to store session data on the client side. Less common in REST APIs, and mostly used in WebApps.
- **Caching -** Improved performance by storing responses.

  NOTE: Use headers to control caching and authentication. Avoid cookies in stateless APIs.

-- STATELESS vs STATEFUL --

- **Stateless API** - It does not retain any information about the client between requests. Each request is independent and must contain all the necessary information for the server to process it.
  - there is no session or context that is stored on the server.
  - the most common protocols are REST, GraphQL (used in microservices and public APIs)
- **Stateful API** - It maintains context or session information between the requests, where the server remembers previous interactions with the client.
  - it requires session management (cookies, tokens, etc.)
  - the most common protocols are SOAP, WebSockets (used in real-time apps, legacy systems)

-- AUTHENTICATION vs AUTHORIZATION --

- **API Keys** - A simple token (usually key) that is passed in the Header or Query. It is good for internal or low-risk APIs and easy to implement.
- Basic Auth - The username and password are encoded in Base64 and sent in the Header.

- **OAuth2** -
  - it is an industry standard protocol for delegated access.
  - it involves access tokens and refresh tokens.
- **JWT (JSON Web Token)** -
  - these are compact, self-contained tokens.
  - it encodes user info and permissions.
  - it is used in stateless authentication.

NOTE: Use OAuth2 or JWT for secure, scalable APIs. API keys are okay for simple use cases.

-- API Authentication Types --

- API Keys
- Basic Auth
- OAuth2.0
- JWT (JSON Web Tokens)
- OAuth2.1 + PKCE
- Mutual TLS (mTLS)
- OpenID Connect (OIDC)
- API Gateway + IAM

-- WEBHOOKS VS WEBSOCKETS --

Feature	Webhooks	WebSockets
Communication Type	One-way (Server → Client)	Two-way (Client ↔ Server)
Trigger Mechanism	Event-driven (push on event)	Persistent connection (real-time exchange)
Connection	No persistent connection	Persistent TCP connection
Use Case	Notifications, automation	Live updates, chat, dashboards
Protocol	HTTP	TCP (via ws:// or wss://)
Latency	Higher (depends on HTTP request)	Low (real-time)
Scalability	Easier to scale	Requires connection management

1b. Logging Concepts

Friday, September 19, 2025 8:40 AM

- Splunk
- Azure Monitor
- AWS CloudWatch
- Prometheus
- ELK Stack (Elasticsearch, Logstash, Kibana)
- Datadog

Feature	Datadog	Splunk	Azure Monitor	CloudWatch	Prometheus	ELK Stack
Type	SaaS	Enterprise Log Analytics	Cloud-native	Cloud-native	Open-source	Open-source
Focus	Full-stack observability	Log & event analytics	Azure services	AWS services	Metrics & alerts	Logs & search
Ease of Setup	Very easy (agent-based)	Moderate	Easy in Azure	Easy in AWS	Manual setup	Manual setup
APM	☑ Yes	❌ Limited	☑ Basic	☑ Basic	❌ No	❌ No
Log Management	☑ Yes	☑ Yes	☑ Yes	☑ Yes	❌ No	☑ Yes
Dashboards	☑ Rich & customizable	☑ Powerful	☑ Native	☑ Native	☑ via Grafana	☑ via Kibana
Security Monitoring	☑ Yes	☑ Yes (SIEM)	☑ Basic	☑ Basic	❌ No	❌ No
Pricing	SaaS, usage-based	Enterprise license	Azure billing	AWS billing	Free	Free

-- Datadog --

- Datadog is a cloud-native monitoring and observability platform that provides -
  - Infrastructure Monitoring
  - Application Performance Monitoring (APM)
  - Log Management
  - Security Monitoring
  - Synthetic Testing
  - Real User Monitoring (RUM)
- It is designed to give full-stack visibility across cloud, on-prem, and hybrid environments.
- Key differentiations of Datadog -
  - Unified Platform: combines metrics, logs, traces, and security in one UI.
  - Cloud-native integrations: 600+ integrations including AWS, Azure, GCP, Kubernetes, Docker, etc.
  - Developer-Friendly: Great for DevOps and SRE teams

1c. Data Concepts

Friday, September 19, 2025 8:52 AM

1d. Security Concepts

Friday, September 19, 2025 8:52 AM

1e. Performance Concepts

Friday, September 19, 2025 8:52 AM

1f. Cloud Concepts

Friday, September 19, 2025 8:56 AM

ML

Wednesday, September 17, 2025 10:01 AM

Model-agnostic tools are methods that can be applied to any machine learning model, regardless of its internal architecture or complexity. This is especially useful for interpreting "black-box" models like neural networks, which are highly accurate but not transparent by design. By ignoring the internal workings of a model and focusing only on its inputs and outputs, model-agnostic tools ensure consistency and enable fair comparisons across different model types.

The below are post-hoc interpretation methods that are applied after a model has been trained.

SHapley Additive exPlanations (SHAP)	**Local Interpretable Model-agnostic Explanations (LIME)**
• **What it does:** Based on cooperative game theory, SHAP calculates the contribution of each feature to a specific prediction. The tool assigns each feature a "Shapley value" that indicates its impact on the model's output. • **Function:** Provides both global feature importance and local explanations for individual predictions. • **Where to find it:** The `shap` Python library.	• **What it does:** Creates a simple, interpretable "surrogate model" (e.g., linear regression) to approximate the black-box model's behavior in the local vicinity of a single prediction. • **Function:** Explains individual predictions by visualizing which features had the most influence, without trying to explain the entire model's behavior. • **Where to find it:** The `lime` Python library.

GOAL

Monday, September 22, 2025 11:03 AM

Data Product Owner

Data Architect

Data PO

Wednesday, September 17, 2025 10:01 AM

◇ Domain & Business Understanding

1. How do you define a data product in the context of insurance or banking?
2. Can you describe a scenario where a data product improved business outcomes?
3. How do you prioritize data features or enhancements for a product roadmap?
4. What KPIs would you track for a data product in underwriting or claims?

◇ Stakeholder Management & Communication

5. How do you handle conflicting priorities between data engineering and business teams?
6. Describe a time when you had to explain a complex data concept to a non-technical stakeholder.
7. How do you gather requirements for a new data product?
8. How do you ensure alignment between data strategy and business goals?

◇ Data Governance & Compliance

9. What steps do you take to ensure data privacy and compliance (e.g., GDPR, HIPAA)?
10. How do you manage data quality issues in a product lifecycle?
11. What role does metadata play in data product management?

◇ Technical & Analytical Acumen

12. How do you work with data scientists and engineers to build scalable data products?
13. What's your experience with data platforms like Azure, AWS, or Snowflake?
14. How do you evaluate the performance of a machine learning model used in a data product?
15. Can you explain the difference between a data lake and a data warehouse?

◇ Agile & Product Management Practices

16. How do you write user stories for data products?
17. What's your approach to backlog grooming for data-related features?
18. How do you measure the success of a sprint focused on data delivery?
19. How do you handle technical debt in data products?

◇ Scenario-Based Questions

20. You're launching a new data product for fraud detection. What steps would you take from ideation to delivery?
21. A dashboard is showing incorrect metrics due to data pipeline issues. How would you resolve it?
22. You're asked to sunset a legacy data product. How would you manage the transition?

Scrum Master

Thursday, October 9, 2025 9:57 AM

Topics to Cover

☑ **1. Scrum & Agile Fundamentals**
- Agile Manifesto & Principles
- Scrum roles: Product Owner, Scrum Master, Development Team
- Scrum ceremonies: Sprint Planning, Daily Scrum, Sprint Review, Retrospective
- Scrum artifacts: Product Backlog, Sprint Backlog, Increment
- Definition of Done vs Acceptance Criteria

☑ **2. Scrum Master Role**
- Servant leadership and coaching mindset
- Conflict resolution and team dynamics
- Removing impediments
- Facilitating Scrum events
- Stakeholder communication

☑ **3. Insurance Domain Knowledge**
- P&C Insurance basics (especially if relevant to Country Financial)
- Claims, underwriting, policy lifecycle
- Regulatory and compliance aspects
- Agile transformation in legacy insurance systems

☑ **4. Agile Tools & Metrics**
- Jira, Confluence, Rally (if applicable)
- Velocity, burndown charts, cumulative flow diagrams
- Agile maturity models
- PI Planning (if SAFe is used)

☑ **5. Behavioral & Situational Questions**
- Handling team conflicts
- Coaching a resistant team
- Working with a difficult Product Owner
- Leading Agile transformation
- Metrics to track team performance

Definition of Done vs Acceptance Criteria

Aspect	Definition of Done	Acceptance Criteria
Scope	Applies to all backlog items	Specific to each user story
Owner	Team-defined	Product Owner-defined
Focus	Quality and Completeness	Business Functionality
Purpose	Ensures consistency and quality	Ensures story meets business needs
Used For	Sprint review, release readiness	Story acceptance and testing

☑ **Definition of Done (DoD) -** A **shared checklist** of activities that must be completed for any product increment (e.g., user story, feature, or sprint) to be considered **"done"**.

- **Characteristics:**
  - Applies **universally** across all user stories or backlog items.
  - Defined by the **team** (often with input from QA, DevOps, PO).
  - Ensures **quality and consistency**.
  - Includes technical and process-related tasks.
- **Examples:**
  - Code is written and peer-reviewed.
  - Unit and integration tests are passed.
  - Documentation is updated.
  - Deployed to staging environment.
  - No critical bugs remain.

☑ **Acceptance Criteria -** A **set of conditions** that a specific user story or feature must meet to be accepted by the **Product Owner**.

- **Characteristics:**
  - **Story-specific** and functional
  - Written by the **Product Owner** or BA
  - Ensures the story delivers **business value**
  - Helps guide **test cases** and validation
- **Examples:**
  - User can submit a claim with all mandatory fields.
  - System sends confirmation email after submission.
  - Error message appears if policy number is invalid.

Metrics

Thursday, October 9, 2025 9:57 AM

-- Metric by Purpose --

- **Delivery Metrics**:
  - **Velocity:** the number of SP's completed per Sprint.
  - **Lead Time:** the time from request to delivery.
  - **Cycle Time:** the time from work start to completion.
  - **Burndown Chart:** a visual representation of remaining Work vs Time.
  - **Work Item Age**: Time a work item has been in progress. Helps identify aging tasks.
  - **Throughput**: Number of work items completed in a given time - this metric complements Velocity

- **Quality Metrics**:
  - **Escaped Defects:** the number of bugs found after the release.
  - **Defect Density:** the number of defects relative to the size of the work product (0.25 defects per SP)

- **Predictability Metrics**:
  - **Team Predictability:** it shows how reliably a team delivers on its planned work during a Program Increment (PI)
  - **PI Objectives Achievement:** it shows how well a team or ART (Agile Release Train) achieves the business objectives committed during PI Planning

- **Flow Metrics**:
  - **Flow Time:** the time taken for an item to go from start to finish
  - **Flow Efficiency:** the ratio of active work time vs total time

- **Engagement Metrics**:
  - **Team Happiness/Engagement:** it shows the qualitative metric from Retrospectives (track Morale and Burnout Risk)

==========

-- Burndown vs Burnup Chart --

- A Burndown Chart shows how much work remains in a sprint or project over time. It is best used for sprint tracking.
- A Burnup Chart shows how much work has been completed and the total scope over time. It is best used for release planning and scope change visibility.

==========

Team-Level Metrics (Scrum/Kanban Teams)

- **Velocity** - It measures the amount of work a team completes in a sprint (story points).
- **Iteration Burndown** - It tracks progress toward completing committed work in a sprint.
- **Team Predictability** - It shows the ratio of Actual vs Planned story points delivered.
- **Defect Density -** the number of defects relative to the size of the work product (0.25 defects per SP)
- **Story Completion Ratio -** % of committed stories completed in a sprint.

Program-Level Metrics (ART – Agile Release Train)

- **PI Objectives Achievement -** Measures how well teams met their committed PI objectives.
- **Feature Cycle Time -** Time taken from feature definition to delivery.
- **Cumulative Flow Diagram (CFD)** - It shows work in progress across states (e.g., backlog, in progress, done).
- **Program Predictability Measure** - Aggregated predictability across all teams in the ART.
- **Dependency Tracking** - Number and impact of inter-team dependencies.

Portfolio-Level Metrics

- **Lean Portfolio Metrics**
  - **Epic Lead Time**: Time from Epic creation to completion.
  - **Epic Success Rate**: % of Epics delivering expected business outcomes.
  - **Investment vs Value Delivered**: Tracks ROI across strategic themes.
- **Strategic Alignment** - It measures how well work aligns with business goals and OKRs (Objectives and Key Results).
- **WSJF (Weighted Shortest Job First)** - Prioritization metric used in SAFe to maximize economic benefit.
- **Budget vs Actual Spend** - Financial tracking for Epics and initiatives.

DevOps & Release Metrics (DORA)

- **Deployment Frequency** - How often code is deployed to production.
- **Change Lead Time** - Time from code commit to production release.
- **Change Failure Rate** - % of deployments causing failures.
- **Mean Time to Recovery (MTTR)** - Time taken to restore service after a failure.
- **Other Considerations -**
  - **Release Frequency Trends**: Helps visualize improvement over time.
  - **Automation Coverage**: % of test cases automated. Supports CI/CD maturity.

==========

SAFe 6.0 emphasizes **Flow Metrics** to improve system-level agility

- **Flow Velocity -** Number of completed items over time
- **Flow Time -** Time taken for an item to go from start to finish
- **Flow Efficiency -** Ratio of active work time vs total time
- **Flow Load -** Number of items in progress
- **Flow Distribution -** Types of work (features, defects, risks) being done
- **Other Considerations -**
  - **Flow Predictability**: Measures how consistent delivery is over time.
  - **Flow Capacity**: Available capacity vs actual usage across teams.

==========

OKRs stand for **Objectives and Key Results**. It's a goal-setting framework used by organizations to align teams and individuals with strategic priorities and measure progress toward outcomes.

- A **qualitative goal** that defines *what* you want to achieve. It should be Inspirational, Time-bound, Clear and Concise.
- A **quantitative measures** that define *how* you'll know you've achieved the objective. It should be Specific & Measurable.

OKR's are used to align teams with business goals, encourage focus on outcomes. improve transparency and accountability, and support Agile and SAFe planning.
OKRs are typically set quarterly, scored between 0.0 and 1.0, and should be cascaded from company to team level.

1. Dependency on Middleware/API Team

Our team relied heavily on the middleware team for API integrations, which often caused sprint delays due to misaligned timelines and unclear ownership.
My Approach:
I facilitated joint planning sessions and introduced a shared dependency board in Jira. We also created synthetic test data using the MAGE tool to simulate API responses, allowing our team to continue development without waiting for production-ready endpoints.

2. Heavy Regression Testing Slowing Releases

We followed a bi-weekly release cadence, but regression testing was consuming significant QA bandwidth, especially with legacy systems involved.
My Approach:
I collaborated with QA leads to implement automated regression suites and prioritized test case refactoring during sprints. We also used synthetic data to avoid HIPAA violations and reduce reliance on masked production data.

3. Tech-Savvy Customer Interfering with Team Dynamics

A stakeholder from the customer side, though well-intentioned, frequently joined team discussions and suggested technical solutions, which disrupted focus and autonomy.
My Approach:
I set clear boundaries by reinforcing the Scrum roles and facilitated stakeholder alignment meetings outside of sprint ceremonies. I also coached the team on how to respectfully manage external inputs while staying focused on sprint goals.

4. Low Agile Maturity & SAFe Adoption

The organization was transitioning to SAFe, but the team lacked clarity on roles and responsibilities, leading to confusion and low morale.
My Approach:
I conducted Agile maturity assessments and tailored coaching sessions for each role—Scrum Team, Product Owner, and Release Train Engineer. I also introduced visual role maps and facilitated team-building retrospectives to rebuild trust and motivation.

Weakness	Growth Strategy
Too detail-oriented	Learned to balance detail with big-picture thinking during sprint reviews.
Hesitant to say "no" to stakeholders	Practiced assertive communication and used data to justify prioritization.
Public speaking nerves	Took Toastmasters and now lead Agile workshops confidently.
Overcommitting to sprint goals	Improved estimation techniques and coached team on sustainable pace.

Collaborated with the **Release Train Engineer (RTE)** to ensure PI planning was effective and that teams committed to realistic objectives.
Facilitated **Scrum of Scrums** twice a week to surface cross-team blockers and dependencies.

SAFe

Monday, October 13, 2025 10:50 AM

SAFe is a framework that helps organizations scale Agile practices across multiple teams. It integrates principles from Agile, Lean, and DevOps to deliver value faster and more predictably.

SAFe = Agile + Lean + DevOps

SAFe operates at four levels:

- Team
- Program (ART - Agile Release Train)
- Large Solution
- Portfolio

As a **Scrum Master** in SAFe, your role expands beyond a single team. You're now a **Team Coach** and a **Servant Leader** within the **Agile Release Train (ART)**.

Key Responsibilities:

- **Facilitate team-level events**: Daily Stand-ups, Iteration Planning, Reviews, and Retrospectives.
- **Support PI Planning**: Help your team prepare for and participate in **Program Increment (PI) Planning**, a key SAFe event held every 8–12 weeks.
- **Coach Agile practices**: Ensure the team follows SAFe Lean-Agile principles.
- **Remove impediments**: Collaborate with other Scrum Masters and the **Release Train Engineer (RTE)** to resolve cross-team blockers.
- **Coordinate with other teams**: Participate in **Scrum of Scrums** to align with other teams on the ART.

Key SAFe Events:

PI Planning	Facilitate team breakout sessions, help estimate capacity
Scrum of Scrums	Represent your team, raise blockers
System Demo	Ensure team delivers demo-ready features
Inspect & Adapt	Facilitate team retrospectives

Key SAFe Artifacts:

Program Backlog	Features prioritized for the ART.
Team Backlog	Stories derived from features.
PI Objectives	Goals for the PI, created by each team.
Program Board	Visualizes features, dependencies, and milestones across teams.

Tips to be a SAFe Scrum Master:

- **Think beyond the team**: Your impact is at the program level.
- **Master facilitation**: PI Planning and Inspect & Adapt are high-stakes events.
- **Promote collaboration**: Foster alignment across teams and stakeholders.
- **Use metrics wisely**: Velocity, predictability, and flow efficiency are key.
- **Champion Lean-Agile mindset**: Be the cultural change agent.

FAQ

Monday, October 13, 2025 11:15 AM

Difference between DOR and DOD

Definition of Ready (DOR)	Definition of Done (DOD)
DoR refers to the criteria that a **user story or backlog item must meet before it can be pulled into a sprint**. It ensures that the team has enough clarity and context to start working on the item.	**DoD** is a **shared understanding of what it means for work to be complete**. It applies to user stories, features, or even entire increments.
• It is focused before the work starts. • It focuses on preparedness. • It ensures clarity and feasibility. • It is owned by PO + Team.	• It is focused after the work is completed. • It focuses on completion. • It ensures quality and completeness. • It is owned by Development Team + QA + PO.
☑ **Typical DoR Criteria:** • Clearly defined **user story** with acceptance criteria • **Dependencies** identified and resolved • **Effort estimation** completed • **Business value** understood • Team has the **capacity** to take it on	☑ **Typical DoD Criteria:** • Code is written, reviewed, and merged • Unit and integration tests passed • Documentation updated • Accepted by Product Owner • Deployed to staging or production (depending on scope)
📌 **Purpose:** • Prevents starting work on vague or incomplete items • Improves sprint planning accuracy • Reduces rework and confusion	📌 **Purpose:** • Ensures consistent quality • Aligns team expectations • Helps track progress accurately