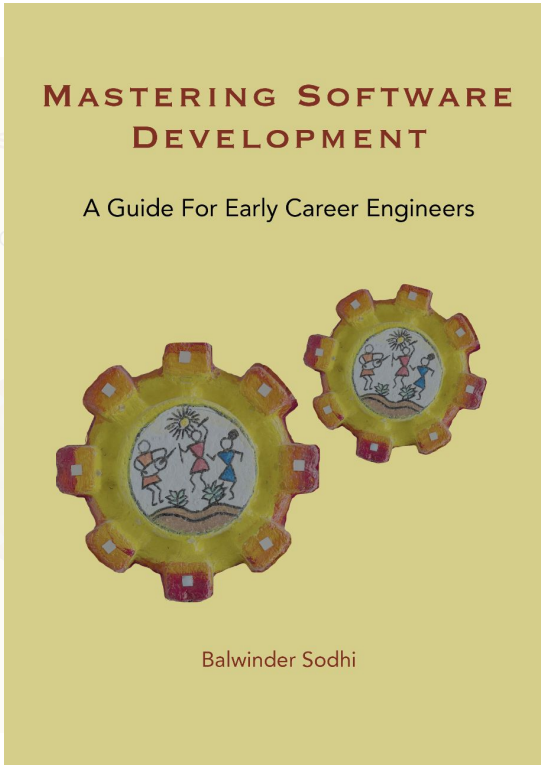


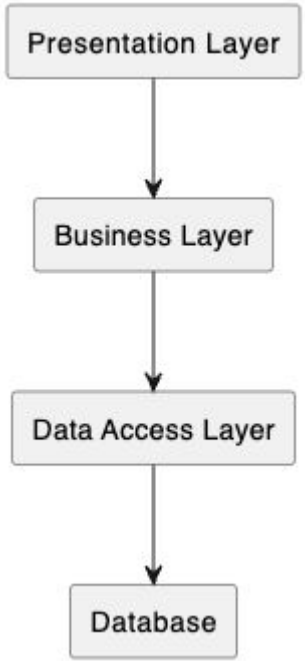
Architectural Design

Balwinder Sodhi

Overview of Major Architectural Styles

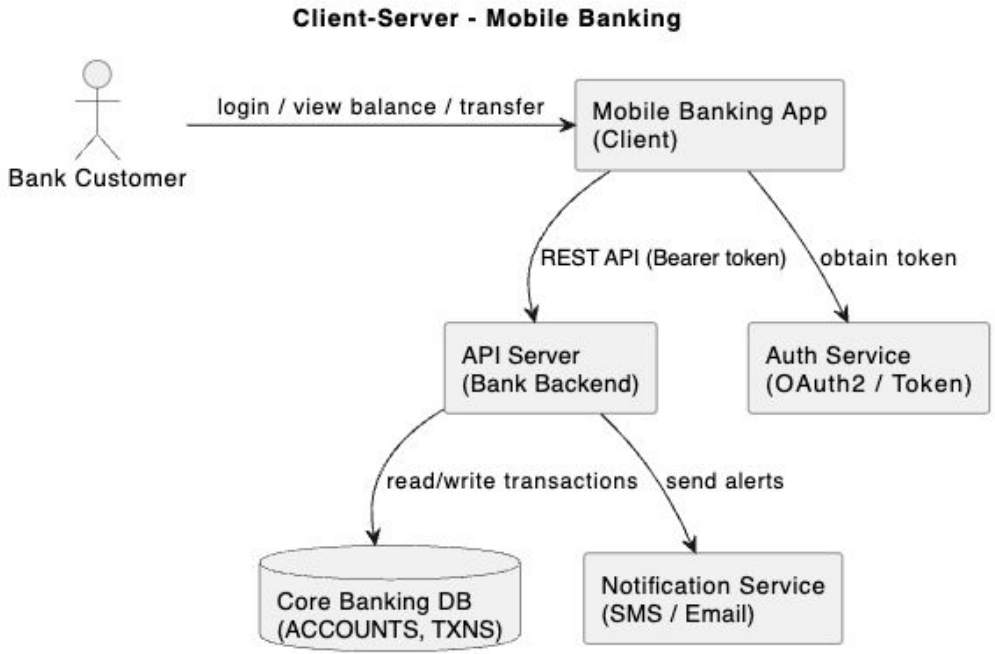
Layered (N-Tier) Architecture

- Key Ideas
  - System divided into layers with distinct responsibilities.
  - Common layers: Presentation, Application/Domain, Data Access.
  - Strict or relaxed layering depending on allowed dependencies.
- Why It's Used
  - Clean separation of concerns.
  - Easier maintainability and testing.
  - Ideal for CRUD + business logic apps.
- Typical Pitfalls
  - Layers become anemic (pure pass-through).
  - Too much coupling when layers leak details.
  - Inefficient for high-performance paths.



Client-Server Architecture

- Key Ideas
  - Clients request services; servers provide them.
  - General pattern behind web, distributed, networked apps.
- Strengths
  - Centralized control and data management.
  - Works for synchronous and asynchronous interactions.
- Pitfalls
  - Server becomes bottleneck.
  - Requires scaling strategies: horizontal, caching, load-balancing.



Microservices Architecture

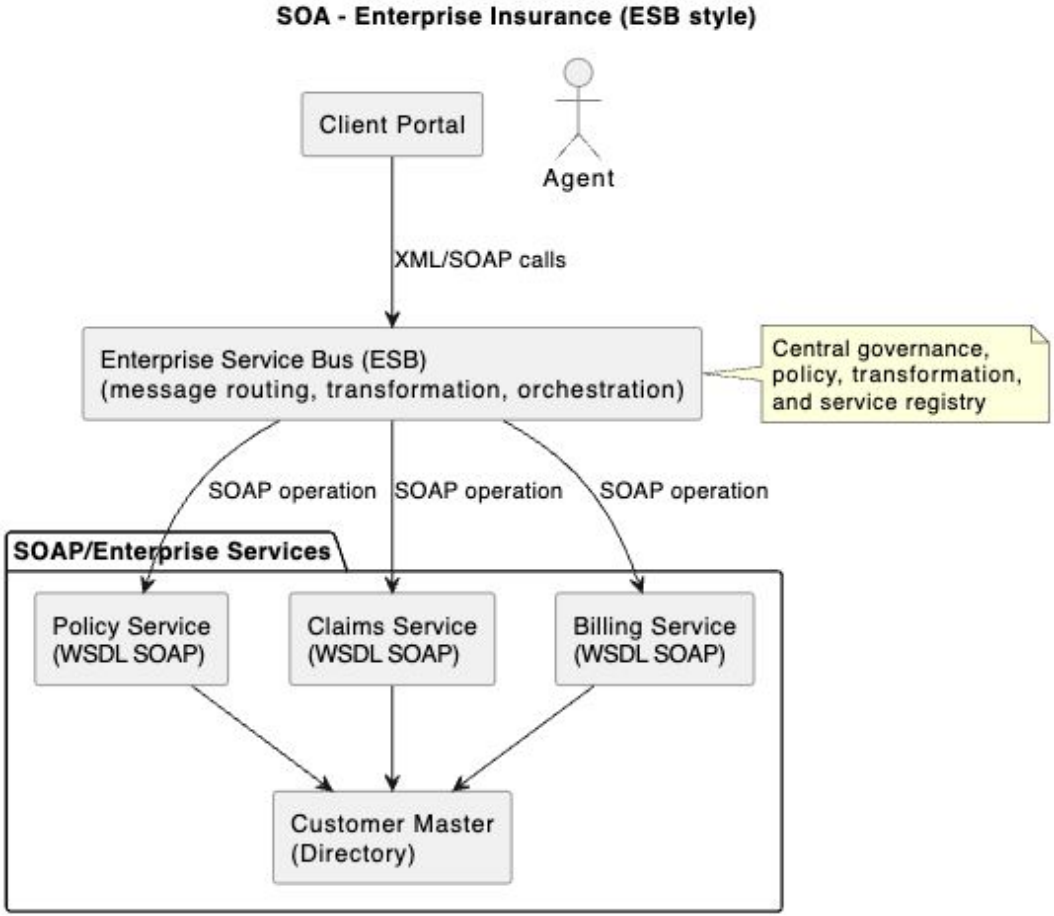
- Key Ideas
  - Independent services communicating over lightweight protocols.
  - Autonomous deployment and scaling.
- Strengths
  - High modularity.
  - Polyglot freedom (services choose tech stack).
  - Independent scaling and fault isolation.
- Pitfalls
  - Distributed system complexity.
  - Network failures, versioning, data consistency issues.
  - DevOps maturity required.

Event-Driven Architecture

- Key Ideas
  - Work triggered by events; decoupled producer–consumer patterns.
  - Supports asynchronous, scalable workflows.
- Strengths
  - Loose coupling.
  - High scalability and resilience.
- Pitfalls
  - Harder debugging and tracing.
  - Eventual consistency issues.

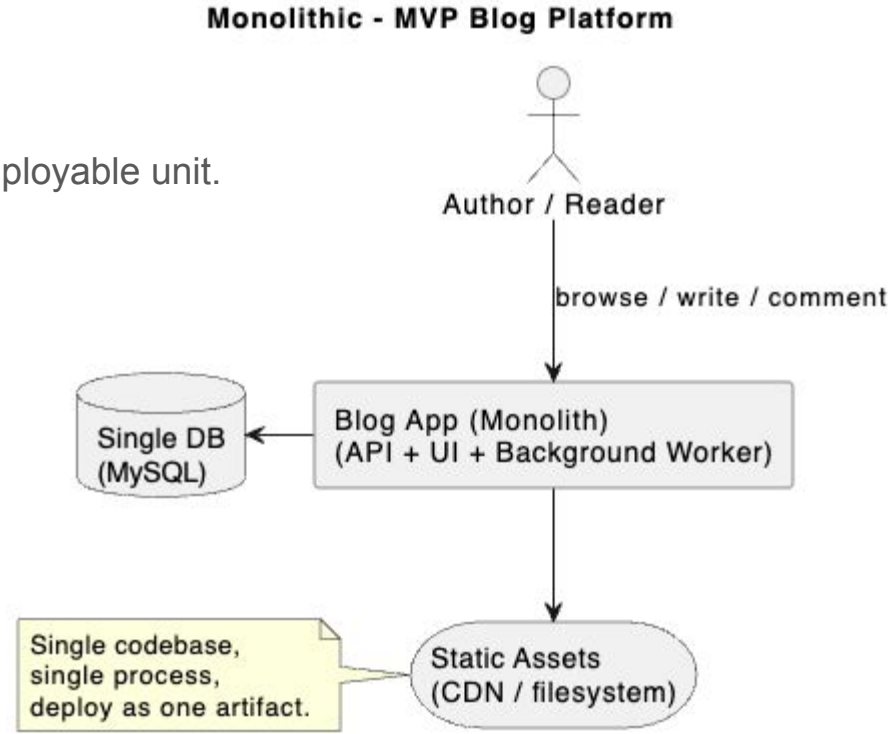
Service-Oriented Architecture (SOA)

- Key Ideas
  - Can be thought of as a predecessor of *micro*services — kind of *macro*services
  - Enterprise services with governance, standardization (SOAP, WS-*).
- Strengths
  - Interoperability across enterprise systems.
  - Strong contract enforcement.
- Pitfalls
  - Heavy tooling.
  - XML/SOAP overhead.


SOA - Enterprise Insurance (ESB style)
Client Portal
Agent
XML/SOAP calls
Enterprise Service Bus (ESB)
(message routing, transformation, orchestration)
Central governance,
policy, transformation,
and service registry
SOAP operation
SOAP operation
SOAP operation
SOAP/Enterprise Services
Policy Service
(WSDL SOAP)
Claims Service
(WSDL SOAP)
Billing Service
(WSDL SOAP)
Customer Master
(Directory)

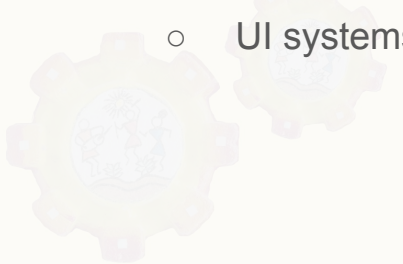
Monolithic Architecture

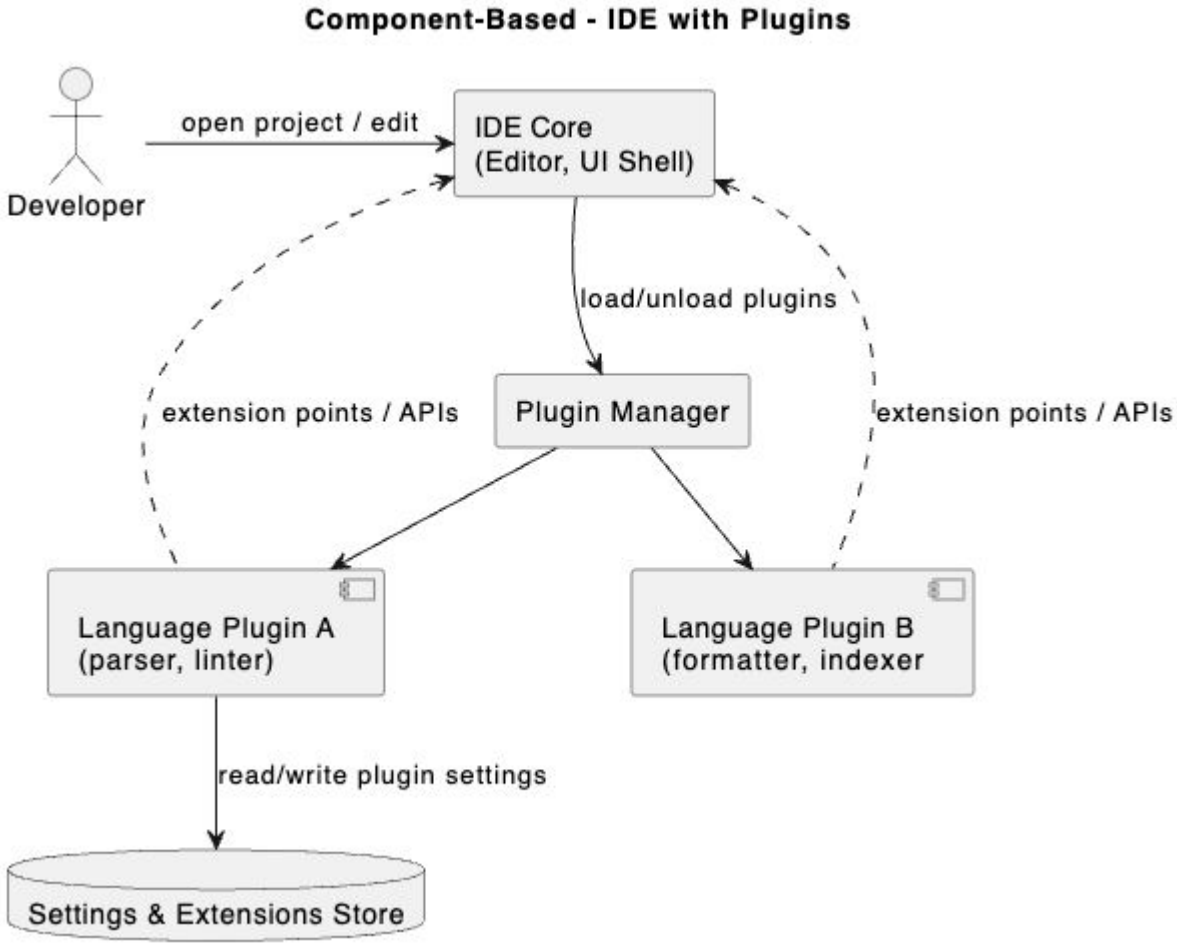
- Key Ideas
  - All functionality deployed as a single deployable unit.
- Strengths
  - Simple development and deployment.
  - Lower operational overhead.
- Pitfalls
  - Slower deployments as app grows.
  - Hard to isolate failures.
  - Scalability is “whole app only”.



Component-Based Architecture

- Key Ideas
  - System composed of reusable, replaceable components.
  - Interfaces define the component contracts.
- Use Cases
  - UI systems, plug-in systems, domain componentization.

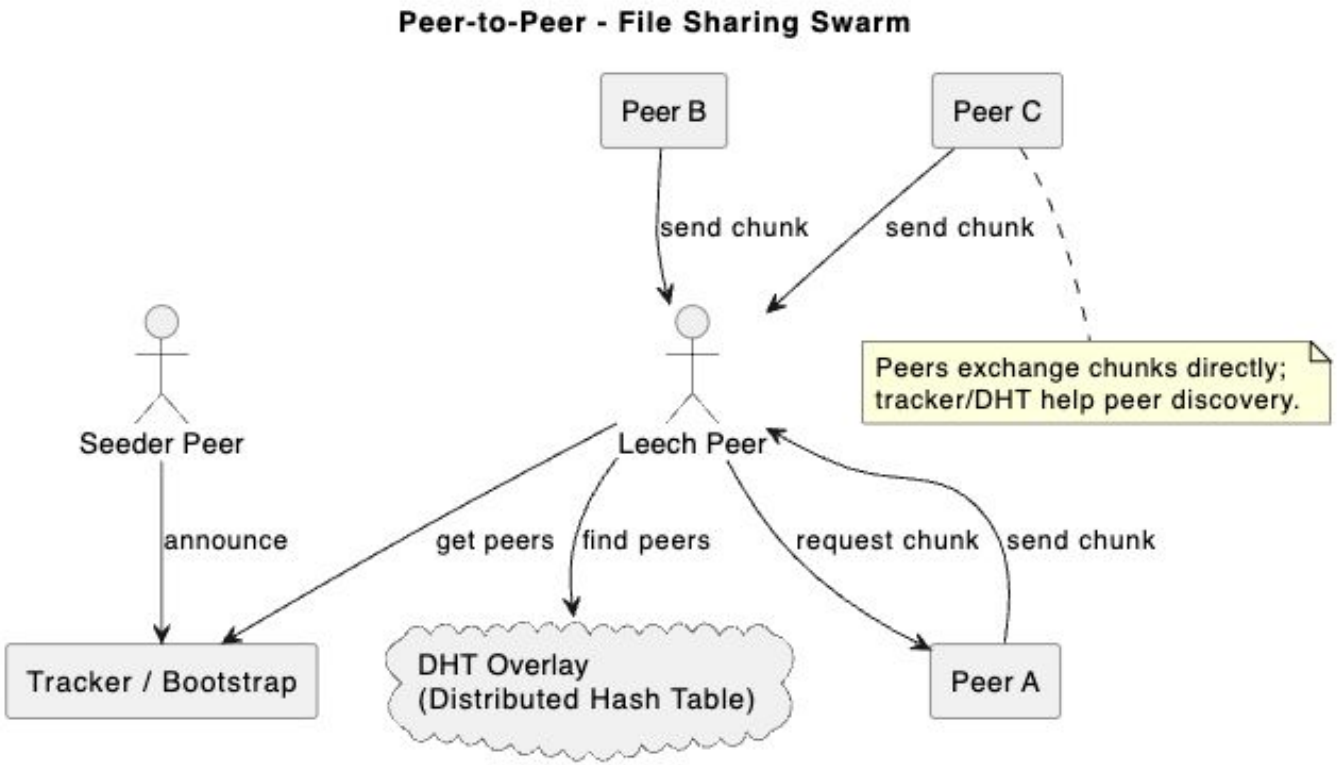



Component-Based - IDE with Plugins
Developer
open project / edit
IDE Core
(Editor, UI Shell)
load/unload plugins
extension points / APIs
Plugin Manager
extension points / APIs
Language Plugin A
(parser, linter)
Language Plugin B
(formatter, indexer
read/write plugin settings
Settings & Extensions Store

Peer-to-Peer (P2P) Architecture

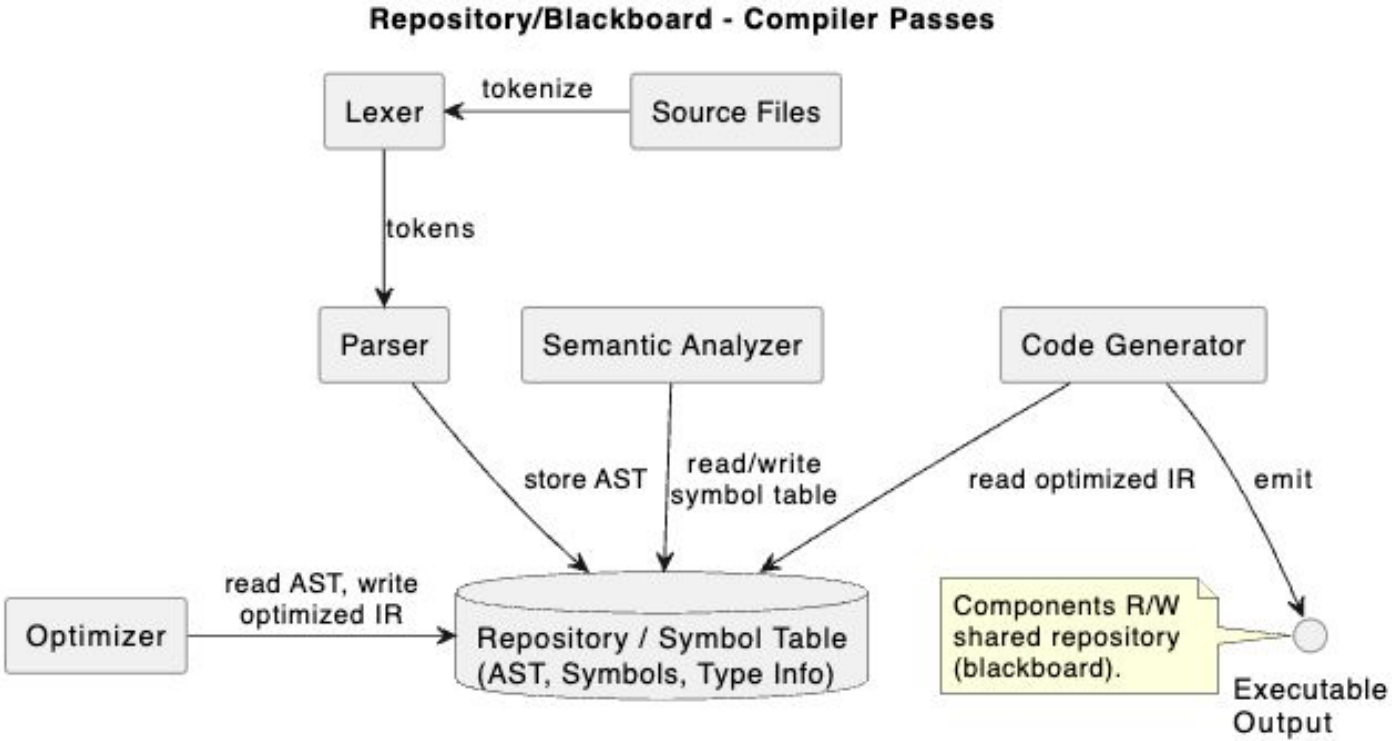
- Key Ideas
  - Nodes act as both clients and servers.
  - Resource sharing without a central authority.
- Use Cases
  - File sharing networks.
  - Distributed ledgers.
  - Decentralized collaboration tools.

Go to in-depth details


Peer-to-Peer - File Sharing Swarm
Peer B
Peer C
send chunk
send chunk
Peers exchange chunks directly;
tracker/DHT help peer discovery.
Seeder Peer
Leech Peer
announce
get peers
find peers
request chunk
send chunk
Tracker / Bootstrap
DHT Overlay
(Distributed Hash Table)
Peer A

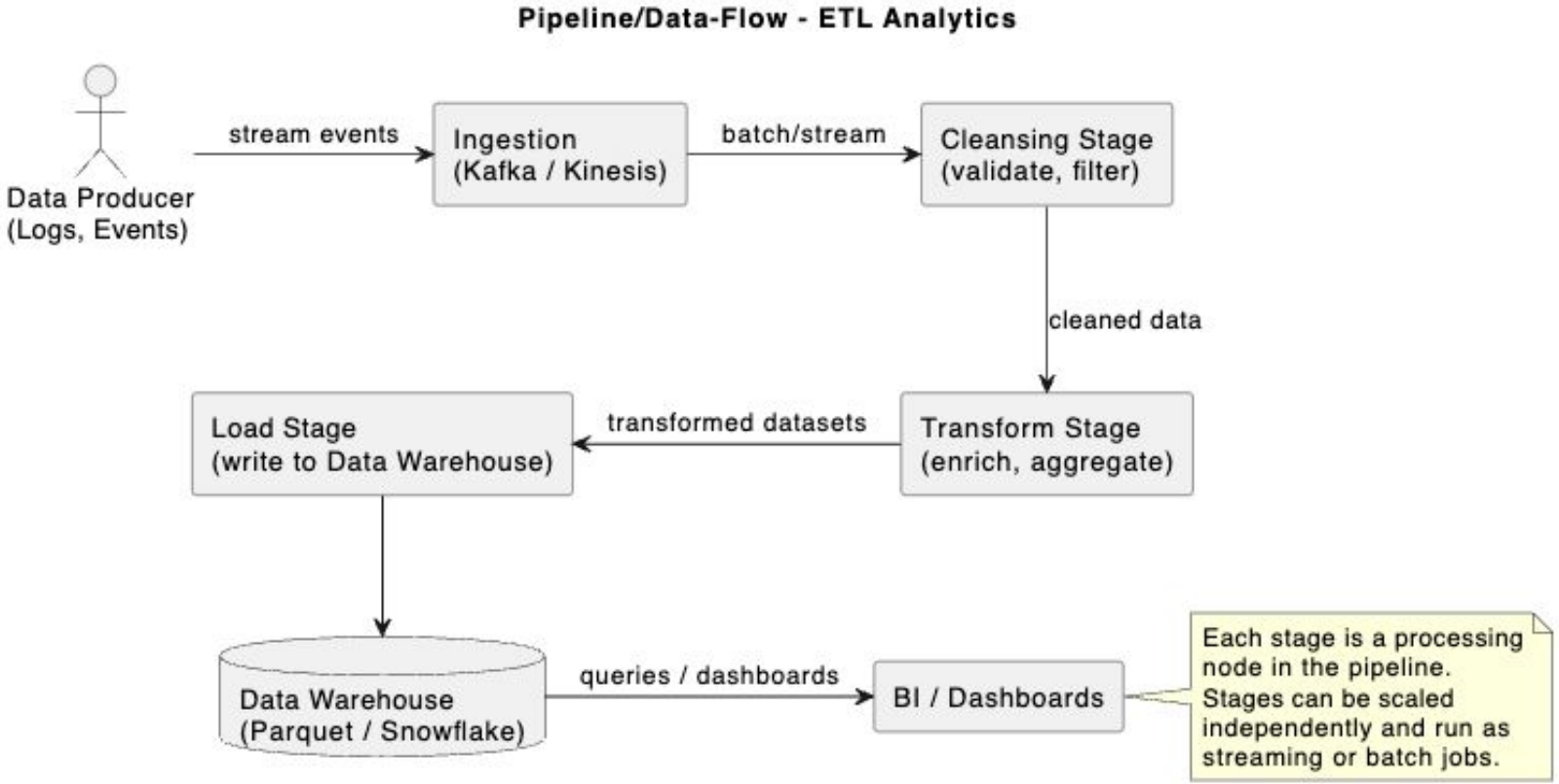
Repository / Blackboard Architecture

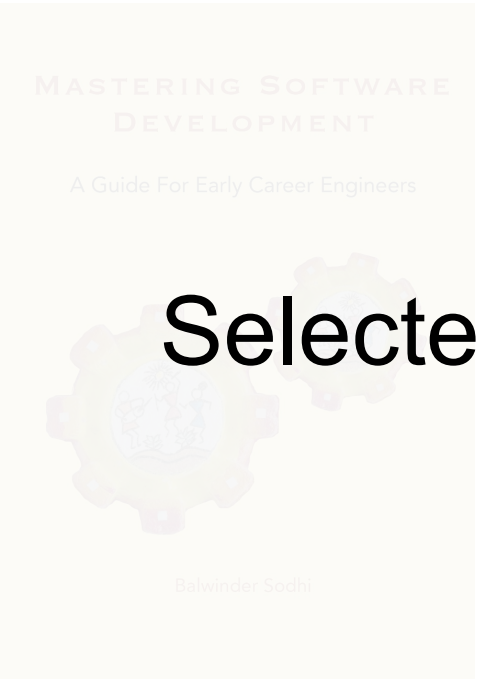
- Key Ideas
  - Shared knowledge base accessed by multiple independent components.
  - Used when you need incremental, cooperative problem solving.
- Examples
  - Compilers (symbol table + passes).
  - AI expert systems.


Repository/Blackboard - Compiler Passes
Lexer
tokenize
Source Files
tokens
Parser
Semantic Analyzer
Code Generator
store AST
read/write
symbol table
read optimized IR
emit
read AST, write
optimized IR
Optimizer
Repository / Symbol Table
(AST, Symbols, Type Info)
Components R/W
shared repository
(blackboard).
Executable
Output

Pipeline / Data Flow Architecture

- Key Ideas
  - Data flows through a series of processing stages.
  - Each stage transforms data and passes to next.
- Use Cases
  - ETL systems, compilers, ML pipelines.

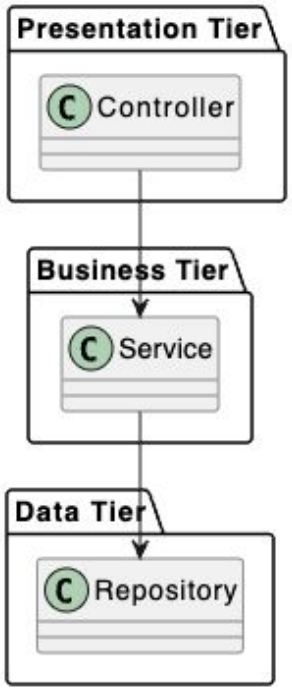

Pipeline/Data-Flow - ETL Analytics
Data Producer
(Logs, Events)
stream events
Ingestion
(Kafka / Kinesis)
batch/stream
Cleansing Stage
(validate, filter)
cleaned data
Transform Stage
(enrich, aggregate)
transformed datasets
Load Stage
(write to Data Warehouse)
Data Warehouse
(Parquet / Snowflake)
queries / dashboards
BI / Dashboards
Each stage is a processing node in the pipeline. Stages can be scaled independently and run as streaming or batch jobs.

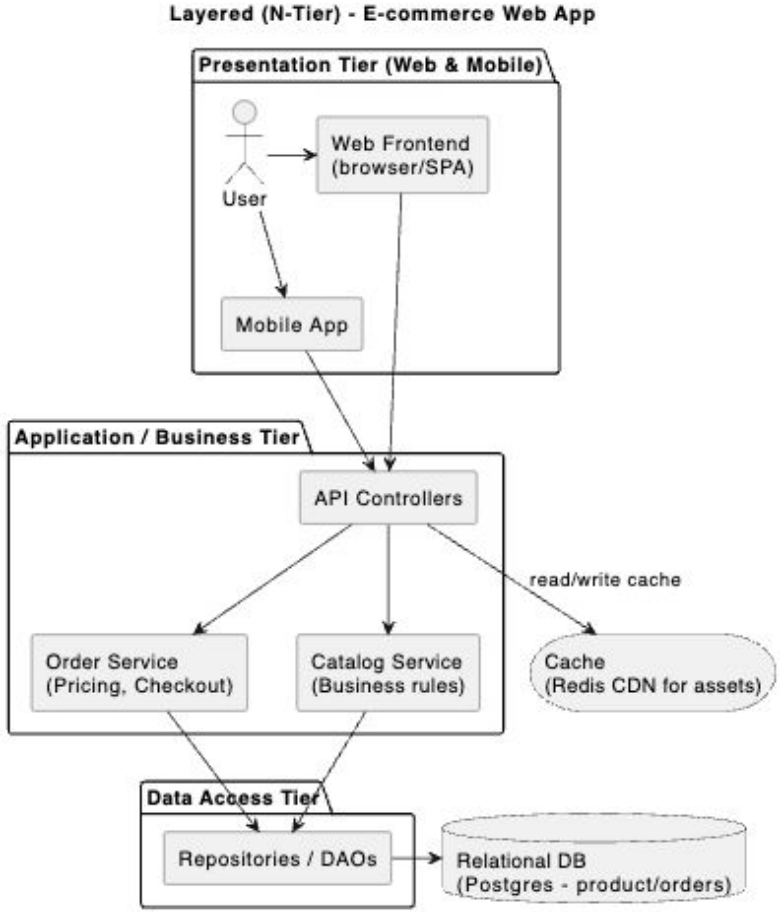


Selected Architectural Styles in Depth

Multi-Tier (N-Tier) Architecture

- Often seen as “simple,” but:
  - the qualities they provide come from certain structural rules
  - these rules directly influence runtime behavior and organizational workflows
- How the Characteristics Emerge
  - Separation of concerns through enforced boundaries
    - Each tier has a single responsibility (UI, domain logic, data operations).
    - Enforced via directory structures, interfaces, or physical deployment boundaries.
    - It reduces accidental coupling because developers cannot easily “reach around” layers.
  - … continued




Layered (N-Tier) - E-commerce Web App
Presentation Tier (Web & Mobile)
User
Web Frontend
(browser/SPA)
Mobile App
Application / Business Tier
API Controllers
read/write cache
Order Service
(Pricing, Checkout)
Catalog Service
(Business rules)
Cache
(Redis CDN for assets)
Data Access Tier
Repositories / DAOs
Relational DB
(Postgres - product/orders)

Multi-Tier (N-Tier) Architecture

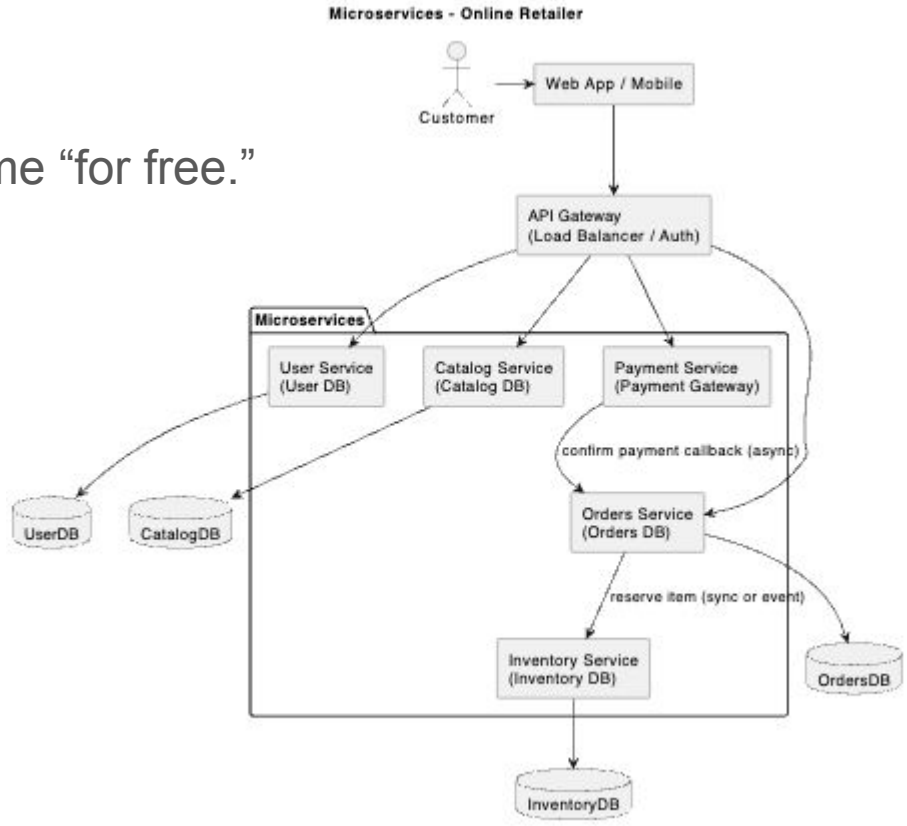
- How the Characteristics Emerge
  - Controlled data flow
    - All data typically flows top → down for reads and writes.
    - This gives predictable call chains and simplifies debugging.
  - Stateless middle tiers
    - Business logic often designed stateless so instances can be replicated behind load balancers.
    - Statelessness → horizontal scaling emerges almost naturally.
  - Shared DB provides consistency
    - A single relational DB ensures strong consistency for most workloads.
    - The architecture itself encourages ACID-style data management.

Multi-Tier (N-Tier) Architecture: How Quality Attributes Are Achieved

- Maintainability
  - Because dependencies flow downward, changes in lower tiers don't break upward layers.
  - Replaceability of layers (e.g., moving from SQL to NoSQL) becomes possible if contracts are stable.
- Security
  - Sensitive components (DB, secrets) reside in inner network zones.
  - Network policies create "security rings" around critical resources.
- Performance
  - Caching can be placed strategically:
    - UI cache (CDN)
    - Application-tier local cache
    - Database read replicas
- Availability
  - Stateless tiers let you scale out horizontally.
  - With load balancers, you gain resilience against instance failures.

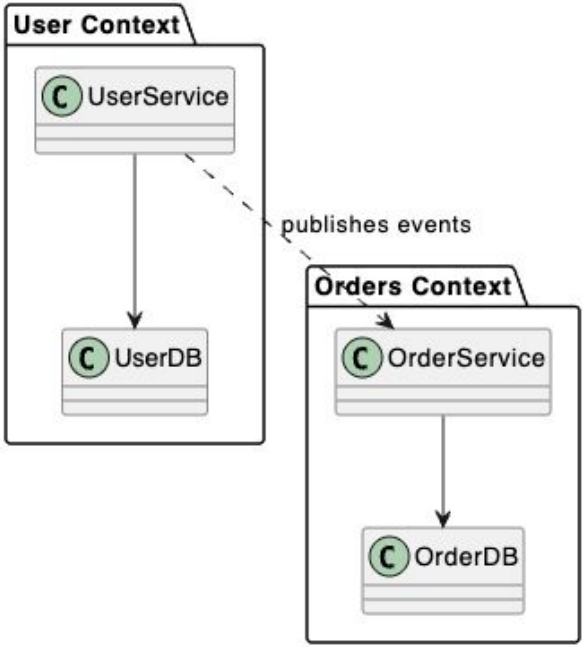
Microservices Architecture

- Microservices qualities do not come “for free.” They arise from:
  - bounded contexts
  - independent deployability
  - distributed data, and
  - asynchronous communication.



Microservices — How the Characteristics Emerge

- Bounded contexts enforce modularity
  - Each service owns a clear, well-defined domain boundary.
  - This prevents business logic bloat in shared modules.
  - Teams work independently without merging conflicts.
- Independent deployment through process isolation
  - Each service is its own deployable unit.
  - CI/CD per service → agility and parallel development.
  - Rolling updates and canary releases reduce downtime risk.
- Distributed data autonomy
  - Each service owns its storage (polyglot persistence).
  - Eliminates shared DB bottlenecks and schema conflicts.
- Communication via APIs and events
  - Services interact through REST/gRPC for sync calls.
  - Events reduce direct coupling and support eventual consistency.

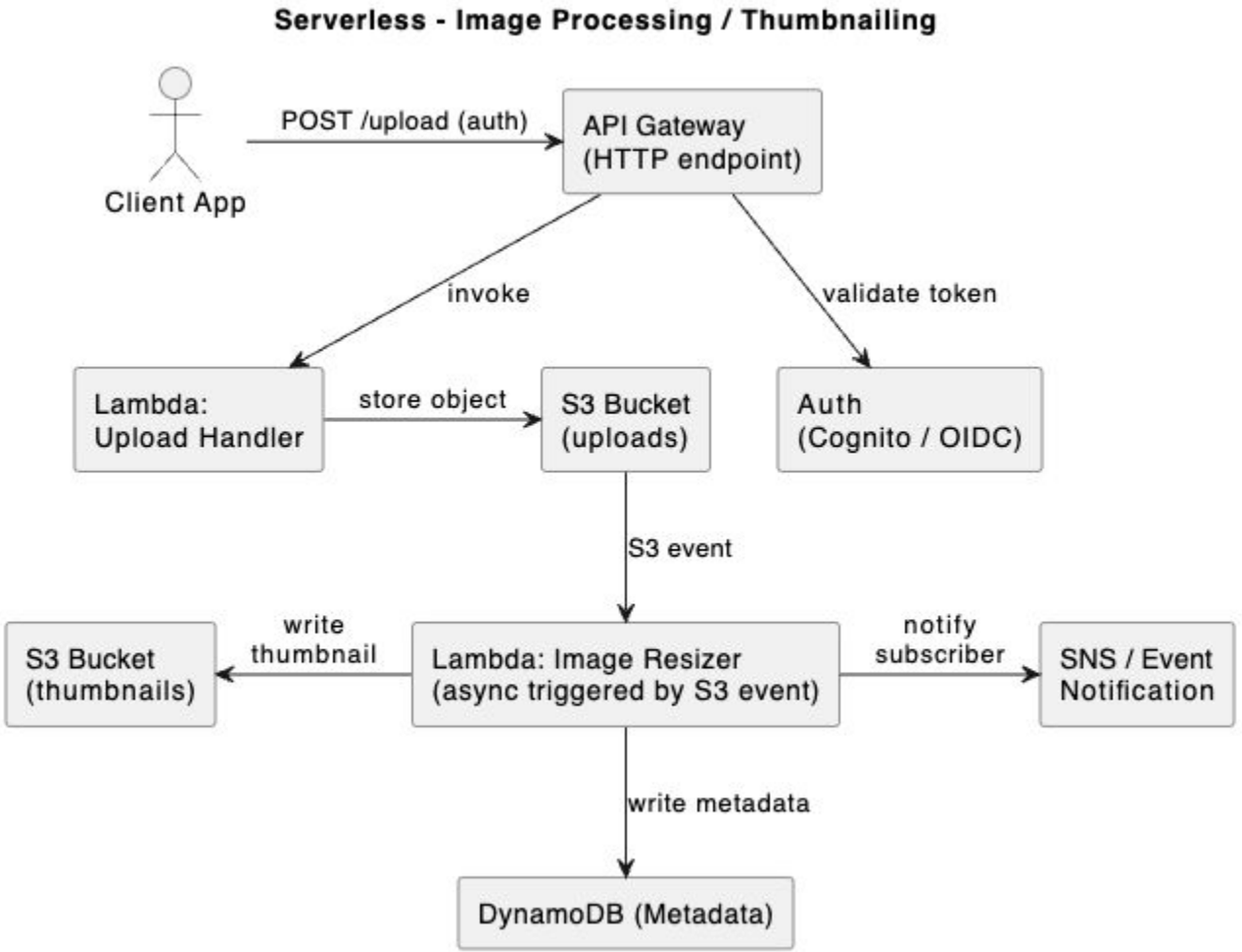


Microservices — How Quality Attributes Are Achieved

- Scalability
  - Services that experience high load (e.g., checkout) can be scaled independently.
  - Hot paths can use faster protocols (gRPC, binary).
- Resilience
  - Failures stay contained:
    - Bulkheads (resource isolation)
    - Circuit breakers
    - Retry/backoff patterns
  - Event-driven communication prevents cascading failures.
- Modifiability
  - A service can switch frameworks, languages, or storage without impacting others.
  - Backward-compatible APIs allow safe evolution over time.
- Deployability
  - Small size → fast build and deploy times.
  - Per-service pipelines eliminate global coordination.

Serverless Architecture

- Serverless qualities arise from:
  - event-driven compute
  - managed runtime environments, and
  - auto-scaling provided by the cloud provider.
- Functions triggered by events
  - Code runs only when invoked by HTTP, queue, cron, or cloud events.
  - Removes the need for dedicated servers or long-lived processes.
- Auto-scaling through ephemeral functions
  - Cloud provider spins up more function instances when traffic spikes.
  - No manual scaling logic to implement.


Serverless - Image Processing / Thumbnailing
Client App
POST /upload (auth)
API Gateway
(HTTP endpoint)
invoke
validate token
Lambda:
Upload Handler
store object
S3 Bucket
(uploads)
Auth
(Cognito / OIDC)
S3 event
S3 Bucket
(thumbnails)
write
thumbnail
Lambda: Image Resizer
(async triggered by S3 event)
notify
subscriber
SNS / Event
Notification
write metadata
DynamoDB (Metadata)

Serverless Architecture

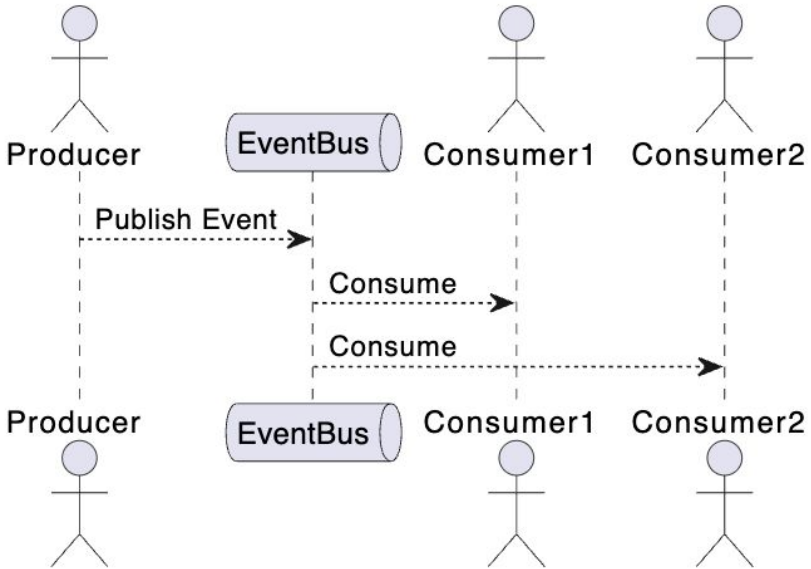
- Fine-grained billing model
  - You pay only for execution time and resource consumption.
  - Idle time costs zero.
- Shared responsibility model
  - Provider handles:
    - OS patching
    - Server provisioning
    - Load balancing
    - Basic security hardening

Serverless Architecture — How Quality Attributes Are Achieved

- Scalability
  - Functions scale horizontally without developer involvement.
  - The platform auto-provisions instances.
- Cost Efficiency
  - Short-lived operations avoid overprovisioning.
  - Great for bursty or erratic workloads (IoT events, batch jobs).
- Availability
  - Cloud providers run functions across multiple AZs.
  - No single-instance failures.
- Time-to-Market
  - Teams focus solely on business logic.
  - No need for container, VM, or infrastructure management.

Event-Driven Architecture

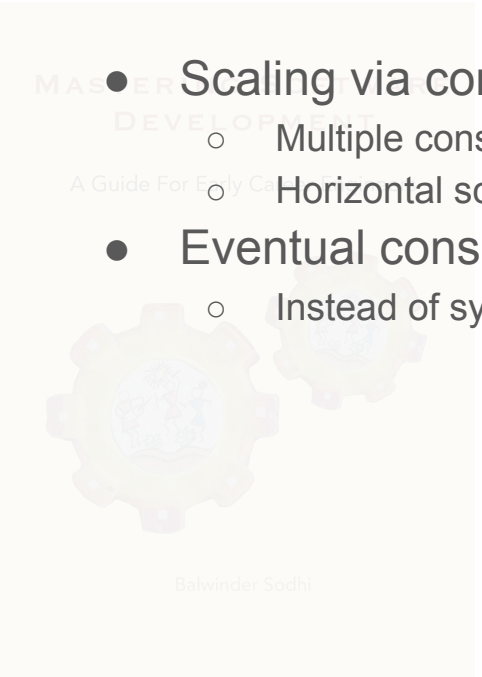
- How the Characteristics Emerge
  - Producers unaware of consumers
    - Producers only publish events; they don’t know who consumes them.
    - New consumers can be added without modifying producers.
  - Event logs as the backbone
    - Systems like Kafka provide:
    - Ordered partitions
    - Durability
    - Replayability



… contd.

Event-Driven: Characteristics — How they emerge

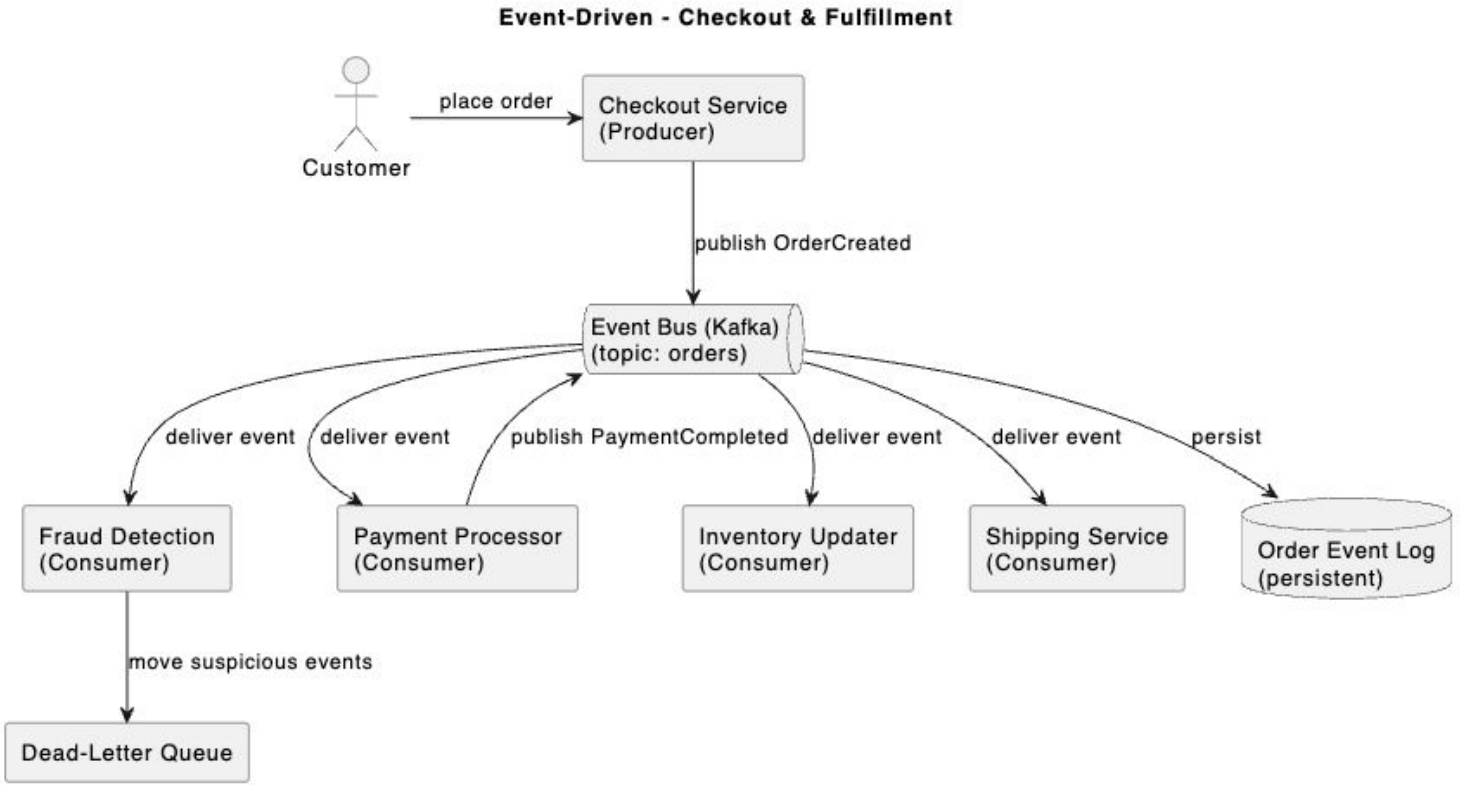
- Scaling via consumer groups
  - Multiple consumers in a group share workload.
  - Horizontal scaling becomes a configuration issue.
- Eventual consistency
  - Instead of synchronous global transactions, services update their own state after events.



Event-Driven: Quality Attributes

- Loose coupling
  - Changes to consumer logic never require touching producers.
  - Encourages autonomous teams.
- Scalability
  - Event brokers handle massive throughput.
  - Consumer groups allow elastic scaling.
- Resilience
  - Event logs preserve events even during outages.
  - Consumers can resume from last offsets.
- Extensibility
  - Adding new downstream processing is straightforward:
    - Real-time analytics
    - Monitoring
    - Fraud detection
    - Alerts

Event-driven Architecture Example



Peer-to-Peer Architecture

- P2P systems rely on:
  - decentralization
  - distributed state, and
  - peer routing protocols.

We have seen the basic details of P2P earlier

P2P: How the Characteristics Emerge

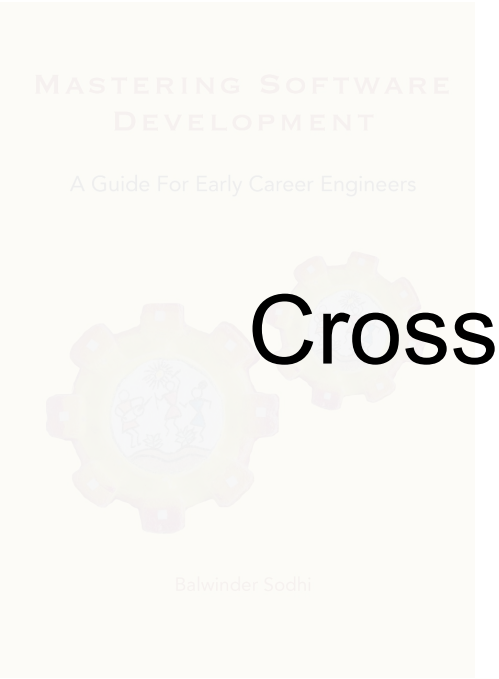
- No central coordinator
  - Peers discover each other using bootstrapping nodes or gossip protocols.
  - Responsibilities are distributed (storage, computation, routing).
- Data replication among peers
  - Replication ensures high availability without servers.
  - Missing data can be reconstructed from neighbor peers.
- Overlay networks (e.g., DHTs)
  - Distributed Hash Tables enable:
    - O(log N) lookup
    - Even distribution of keys
- Self-scaling
  - More peers = more capacity.
  - Aggregate bandwidth and storage grow automatically.

P2P: How Quality Attributes Are Achieved

- Resilience
  - No single point of failure because responsibilities are decentralized.
- Availability
  - Robust replication ensures data survivability.
- Scalability
  - As more peers join, the network gains more resources.
  - Non-linear scaling for file-sharing (BitTorrent swarms).
- Decentralization
  - Useful in censorship-resistant or trust-minimized systems.

Choosing the Right Architecture

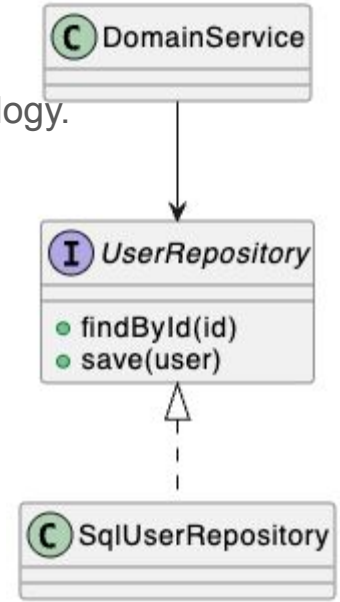
- Key Decision Factors
  - Functional requirements
  - Non-functional goals (latency, throughput, availability)
  - Team skill & organization
  - Operational maturity (DevOps, observability)
  - Regulatory constraints
  - Expected scale & workload shape
- Key Trade-offs
  - Monolith ↔ Microservices
  - Event-Driven ↔ Request/Response
  - Serverless ↔ Containerized
  - Strong consistency ↔ Eventual consistency



Cross-Cutting Architecture Issues

Data Access Abstraction

- Why
  - Reduce coupling between business logic and database schema/technology.
  - Enable database migrations or vendor switches.
  - Improve testability by mocking data access.
- How
  - Repository or DAO pattern hides SQL/ORM specifics.
  - Use interfaces to abstract DB operations.
  - Use mappers to convert between domain and persistence models.



Authentication and Authorization

- Why
  - Identify users and enforce permissions.
  - Required for security, auditability, compliance.
- How
  - OAuth2/OIDC: token-based workflows with identity providers.
  - Access tokens signed using JWTs.
  - Role-based access control (RBAC) via roles → permissions.
  - Attribute-based access control (ABAC) using runtime policies.
- Operational Considerations
  - Token rotation and expiry.
  - Store passwords using slow hashing (bcrypt, Argon2).
  - Enforce MFA, rate-limiting login attempts.

Securing Data in Transit and At Rest

- Why
  - Prevent leakage through network sniffing or disk compromise.
  - Regulatory mandates (GDPR, HIPAA, PCI DSS).
- How
  - TLS 1.2+ for transport; enforce HSTS.
  - Encrypt sensitive fields with AEAD ciphers (AES-GCM).
  - Store keys using KMS or HSM for rotation and access control.
  - Encrypted backups and secure snapshot storage.

Input Validation & Sanitization

- Why
  - Prevent injection attacks.
  - Ensure system receives data in predictable formats.
  - Improves usability
- How
  - Strict schema validation at trust boundaries.
  - Escape untrusted input (HTML escape, SQL bind parameters).
  - Treat all external input as hostile.
  - Use allowlists instead of blocklists.

Logging and Auditing

- Why
  - Observability, security, compliance.
  - Helps troubleshooting and forensic analysis.
- How
  - Use structured logs (JSON).
  - Include correlation IDs to trace multi-service flows.
  - Sensitive-information filters to prevent leaking secrets.
  - Centralized log aggregation via ELK, Loki, or CloudWatch.

Instrumentation and Metrics

- Why
  - Lets teams monitor system health and performance.
  - Detect anomalies early.
- How
  - Use RED (Rate, Errors, Duration) for microservices.
  - Use USE (Utilization, Saturation, Errors) for infrastructure.
  - Export metrics using OpenTelemetry.
  - Add distributed tracing using spans across service boundaries.

Concurrency and Parallelism

- Why
  - Improve performance.
  - Utilize multi-core hardware effectively.
- How
  - Employ multi- threading/processing
  - Worker pools, event-driven design, queues, etc.
- Key considerations
  - Avoid shared mutable state → race conditions.
  - Use messaging or actor model for isolation.
  - Use locks sparingly; favor lock-free structures.
  - Manage thread pools carefully to avoid exhaustion.

Error Handling

- Why
  - Improve reliability.
  - Prevent cascading failures.
  - Enhance user experience.
- How
  - Central error handlers at API boundaries.
  - Retries with jitter to avoid thundering herd.
  - Circuit breakers for remote calls.
  - Error/ dead letter queue handling for failed messages.

API Design & Integration

API Styles Overview

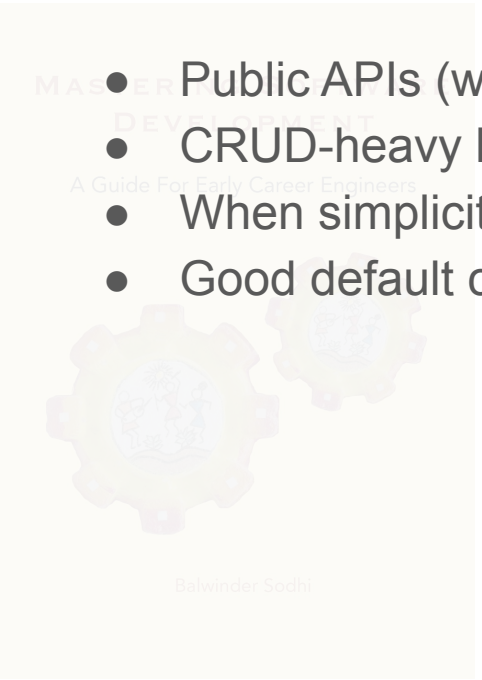
- REST
- SOAP
- GraphQL
- WebSockets
- RPC (gRPC)
- Library/SDK
- File-based
- Custom Protocols
- Serverless APIs

REST: Key Ideas

- Resource-oriented (/users, /orders/123)
- Stateless HTTP interactions
- Uniform verbs: GET, POST, PUT, DELETE
- Uses standard HTTP semantics and caching
- Ideal for CRUD-type operations

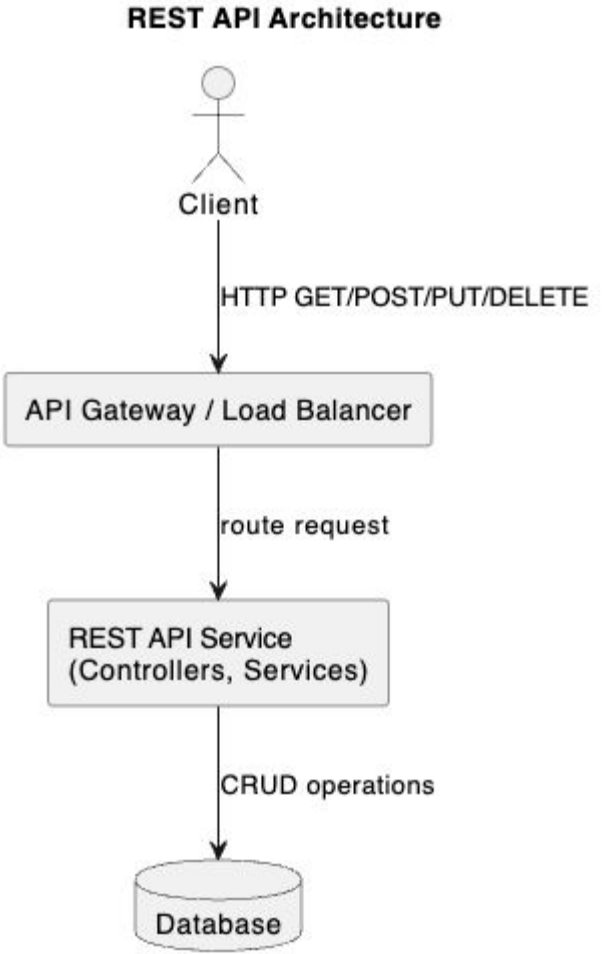
REST: When to Use

- Public APIs (web services, mobile apps)
- CRUD-heavy business systems
- When simplicity, compatibility, browser-friendliness matter
- Good default choice for most teams



REST: How Typically Implemented

- HTTP server exposes resource endpoints
- JSON payloads
- Web clients or mobile apps consume via fetch/axios
- Backed by controllers → services → database
- Uses auth tokens (JWT, OAuth2)



SOAP: Key Ideas

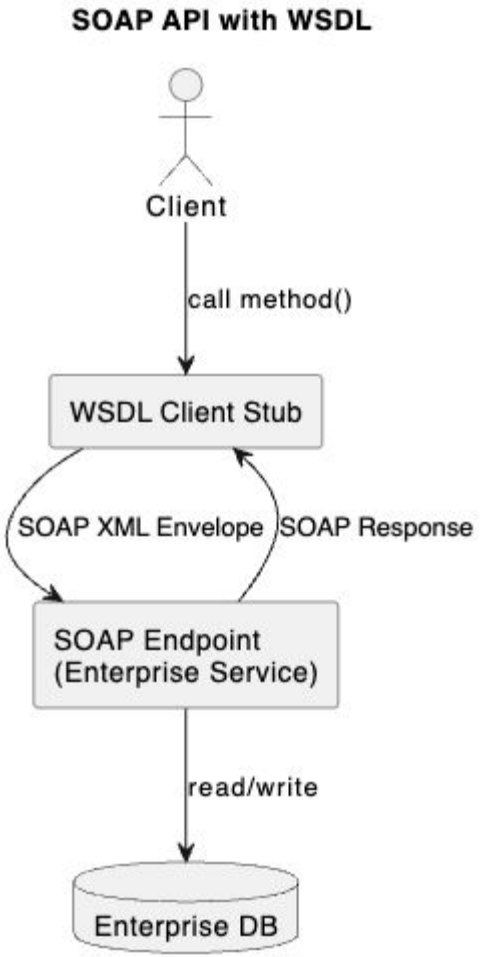
- XML-based protocol
- Strong typing via WSDL (contract-first)
- Heavy on standards: WS-Security, WS-Addressing
- Supports formal enterprise workflows

SOAP: When to Use

- Enterprises with legacy or regulated systems
- Strict schema and validation requirements
- Banking, insurance, telecom
- Systems needing guaranteed message delivery and WS-* standards

SOAP: How Typically Implemented

- SOAP server exposes WSDL contract
- XML messages exchanged over HTTP or Message Queues
- Client stubs auto-generated from WSDL
- Often deployed behind enterprise ESBs

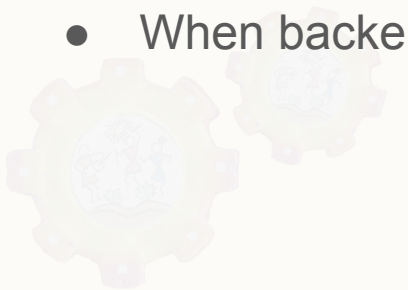


GraphQL: Key Ideas

- Client specifies exactly what data it wants
- Single endpoint (/graphql)
- Strong schema (SDL) defines types
- No over-fetching or under-fetching
- Ideal for modern UI with variable data needs

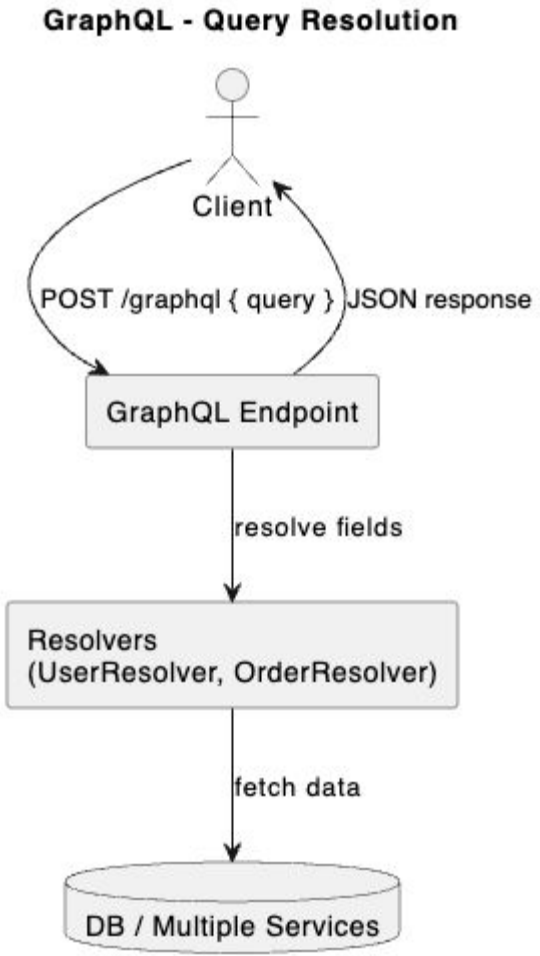
GraphQL: When to Use

- Complex data graphs (social feeds, product catalogs)
- Mobile apps minimizing payload size
- Frontend teams want flexibility
- When backend teams want schema governance



GraphQL: How Typically Implemente

- GraphQL server parses queries → resolvers → services → DB
- Resolvers fetch only required fields
- Supports batching, caching, pagination

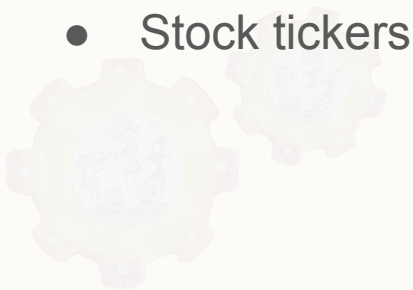


WebSockets: Key Ideas

- Full-duplex, persistent connection
- Real-time bidirectional communication
- No polling needed
- Ideal for continuous streams or updates

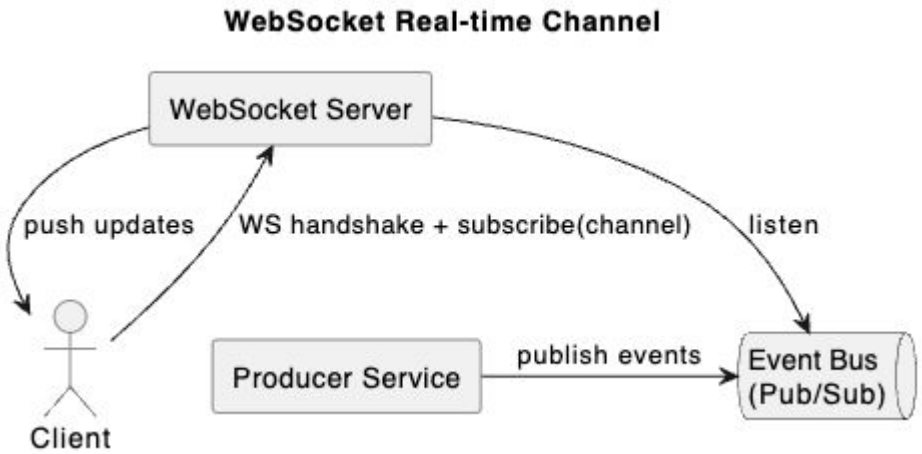
WebSockets: When to Use

- Chat apps
- Multiplayer games
- Real-time dashboards
- Stock tickers, IoT telemetry



WebSockets: How Typically Implemented

- Client upgrades HTTP → WebSocket handshake
- Server and client exchange messages
- Backend services push events to clients via broker or pub/sub system

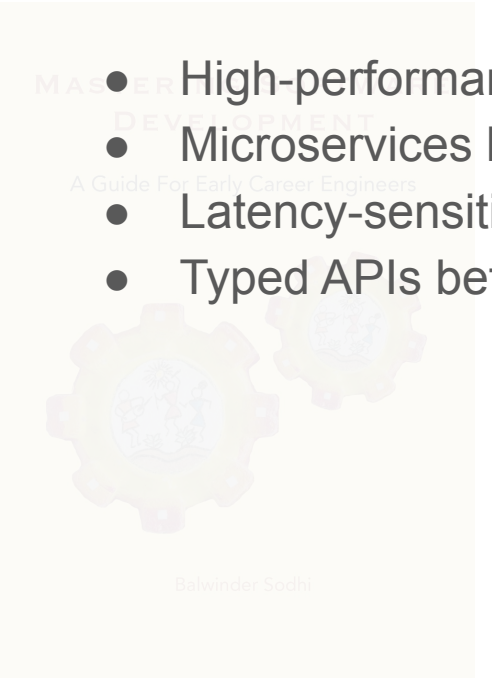


RPC (gRPC): Key Ideas

- Contract-first via Protocol Buffers
- Fast binary messages
- Bi-directional streaming supported
- Strongly typed stubs auto-generated

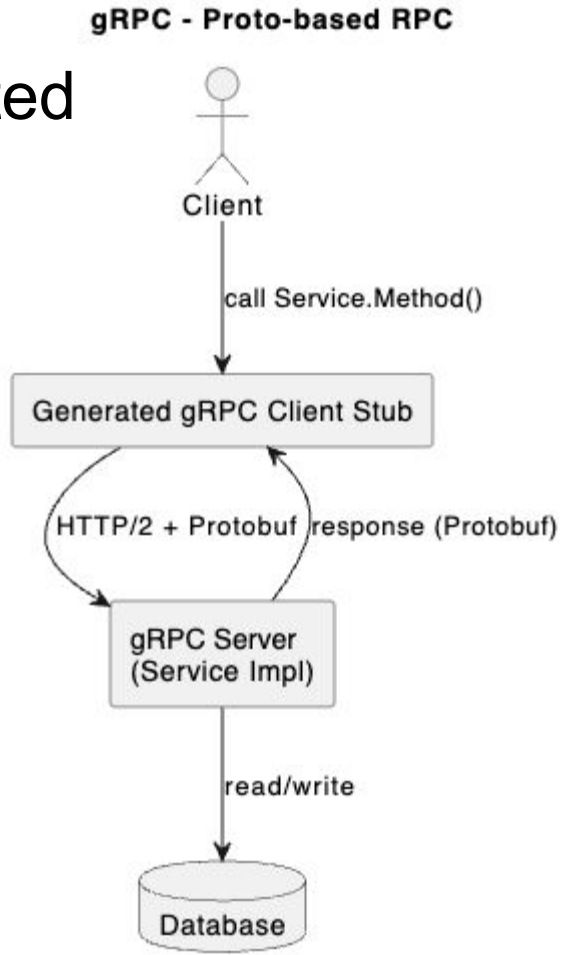
RPC (gRPC): When to Use

- High-performance inter-service calls
- Microservices backend communication
- Latency-sensitive operations
- Typed APIs between trusted services



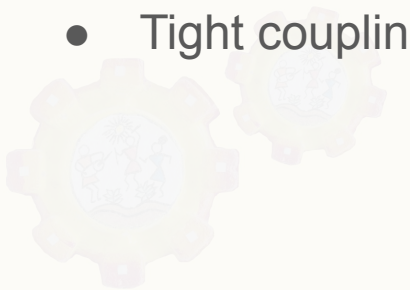
RPC (gRPC): How Typically Implemented

- .proto file defines request/response
- Codegen generates client/server stubs
- Server handles method calls directly
- HTTP/2 transport gives multiplexed streams



Library/SDK APIs: Key Ideas

- Delivered as a code package (npm, pip, Maven, NuGet)
- Runs inside client's application runtime
- Ideal for offline logic, wrappers, utilities
- Tight coupling with client code

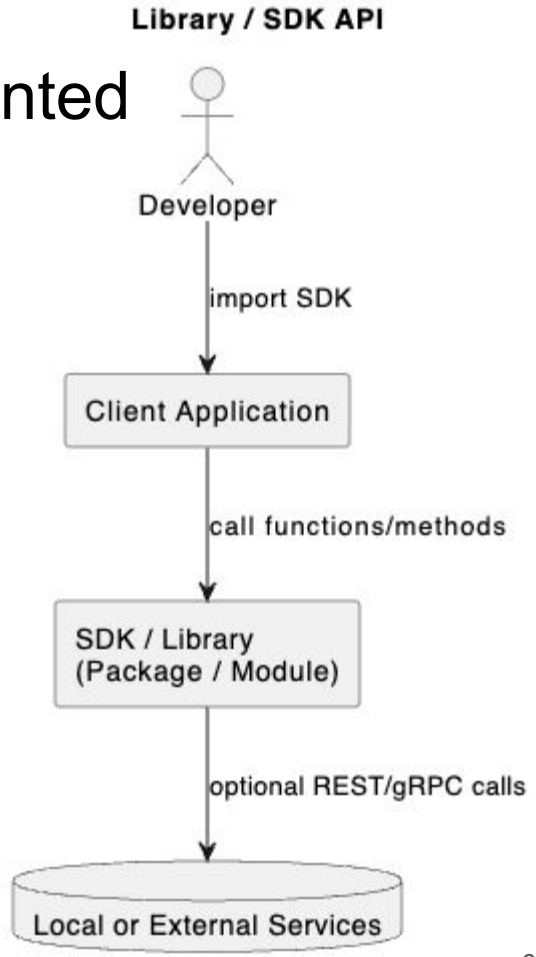


Library/SDK APIs: When to Use

- Domain logic as reusable modules
- Complex integrations wrapped behind easy interfaces
- Performance-sensitive local functionality (crypto, image processing)

Library/SDK APIs: How Typically Implemented

- Maintained as versioned library
- Semantic versioning
- Exposes classes, functions, interfaces
- Client applications import and use directly

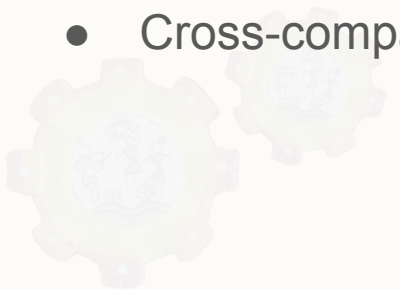


File-Based APIs: Key Ideas

- Integrations via exchanged files (CSV, JSONL, XML, Parquet)
- Scheduled or ad-hoc batch transfers
- Often used for ETL or legacy integrations

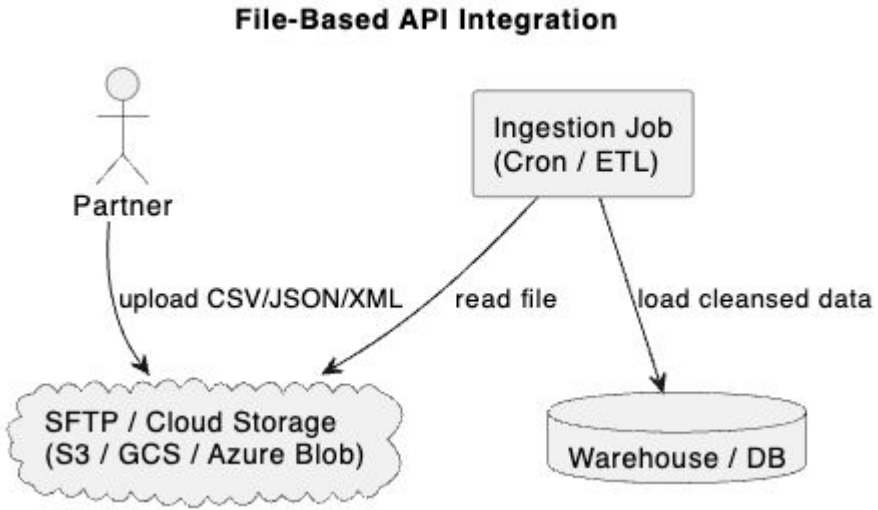
File-Based APIs: When to Use

- Bulk data exchanges
- Partners who cannot expose HTTP APIs
- BI/analytics ingestion
- Cross-company data sharing



File-Based APIs: How Typically Implemented

- Files placed on SFTP, cloud storage, shared bucket
- Scheduled jobs read, validate, transform files
- Errors logged to error bucket or audit trail



Custom Protocols: Key Ideas

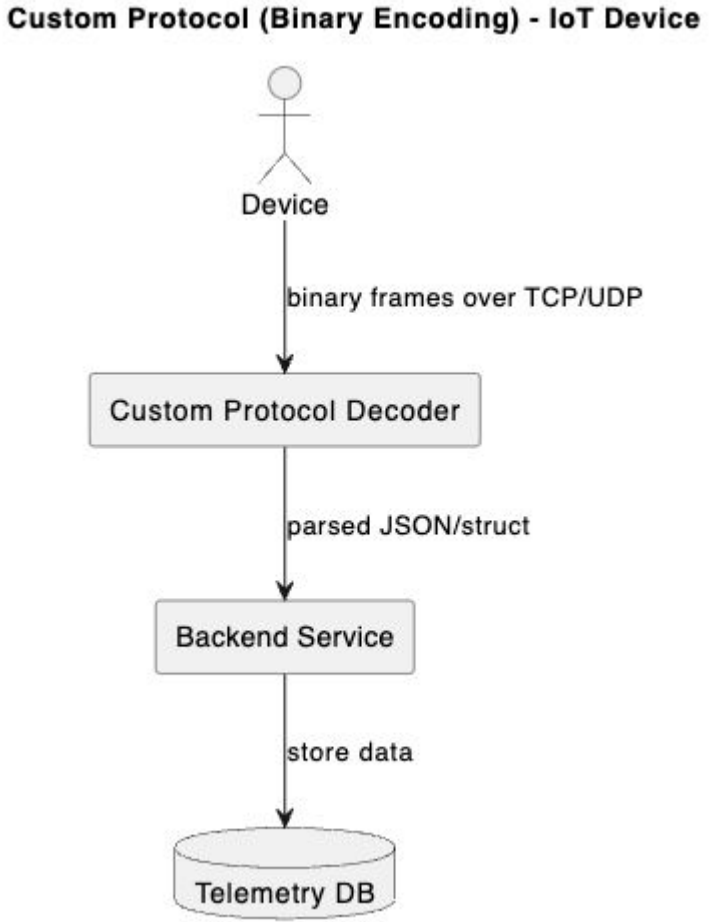
- Built for special constraints (performance, embedded systems, IoT)
- Uses bespoke binary framing, compression, or encoding
- Allows extreme optimization or special hardware interaction

Custom Protocols: When to Use

- Real-time embedded systems
- Low-latency trading systems
- IoT constrained devices
- When standard protocols are too slow or heavy

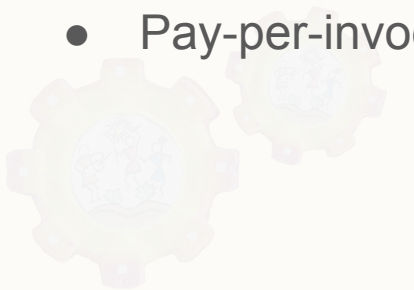
How Typically Implemented

- Custom serialization formats (CBOR, Cap'n Proto, FlatBuffers)
- Own framing rules
- Runs over TCP/UDP/QUIC
- Often includes custom error codes



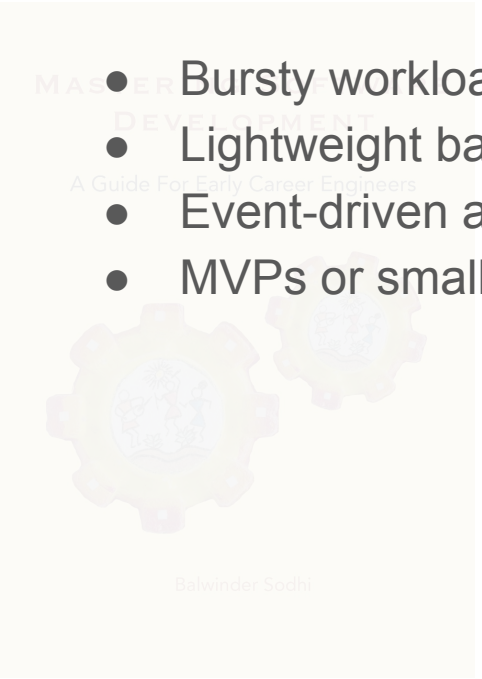
Serverless APIs: Key Ideas

- Functions-as-a-Service invoked by HTTP, events, or schedules
- Zero server management
- Automatic scaling
- Pay-per-invocation cost model



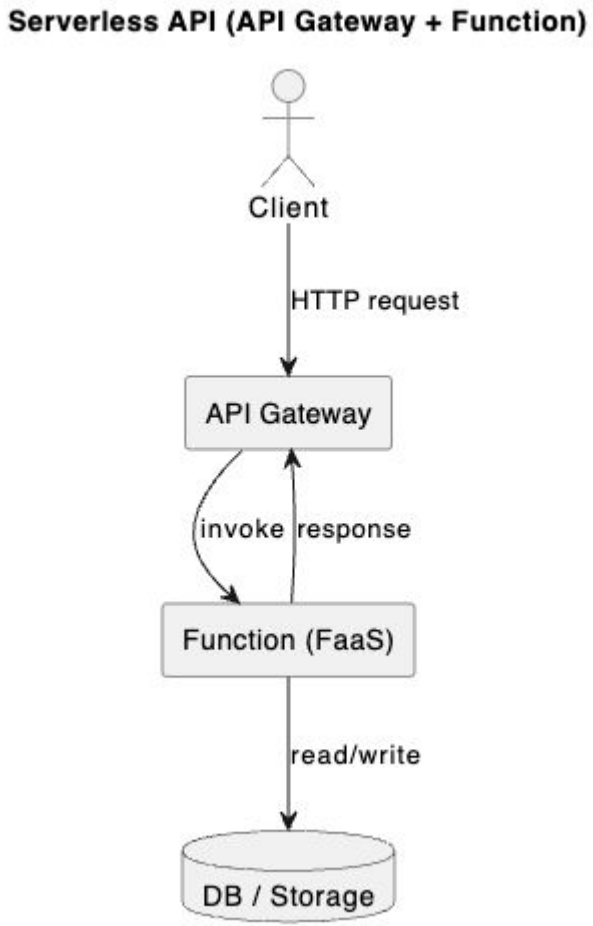
When to Use

- Bursty workloads
- Lightweight backend logic
- Event-driven automation
- MVPs or small teams without DevOps overhead



How Typically Implemented

- API Gateway provides the HTTP endpoint
- Function runs business logic
- S3/DB/Queue integrations via triggers
- Authentication integrated (Cognito/OIDC)



How to Choose an API Style

- REST → simple CRUD, wide client compatibility.
- GraphQL → flexible client-side data needs.
- gRPC → low-latency service-to-service calls.
- WebSockets → real-time updates.
- SOAP → strict enterprise contract governance.
- File-based → batch integrations.
- Serverless → bursty or event-driven workloads.

Overall Summary

- Architectural design is about selecting a structure aligned with business, technical, and operational constraints.
- No single architecture fits all cases; trade-offs matter.
- Styles differ in modularity, scalability, cost, governance, and complexity.
- Cross-cutting concerns must be addressed regardless of chosen architecture.
- API style selection depends on usage patterns, consumers, and performance constraints.