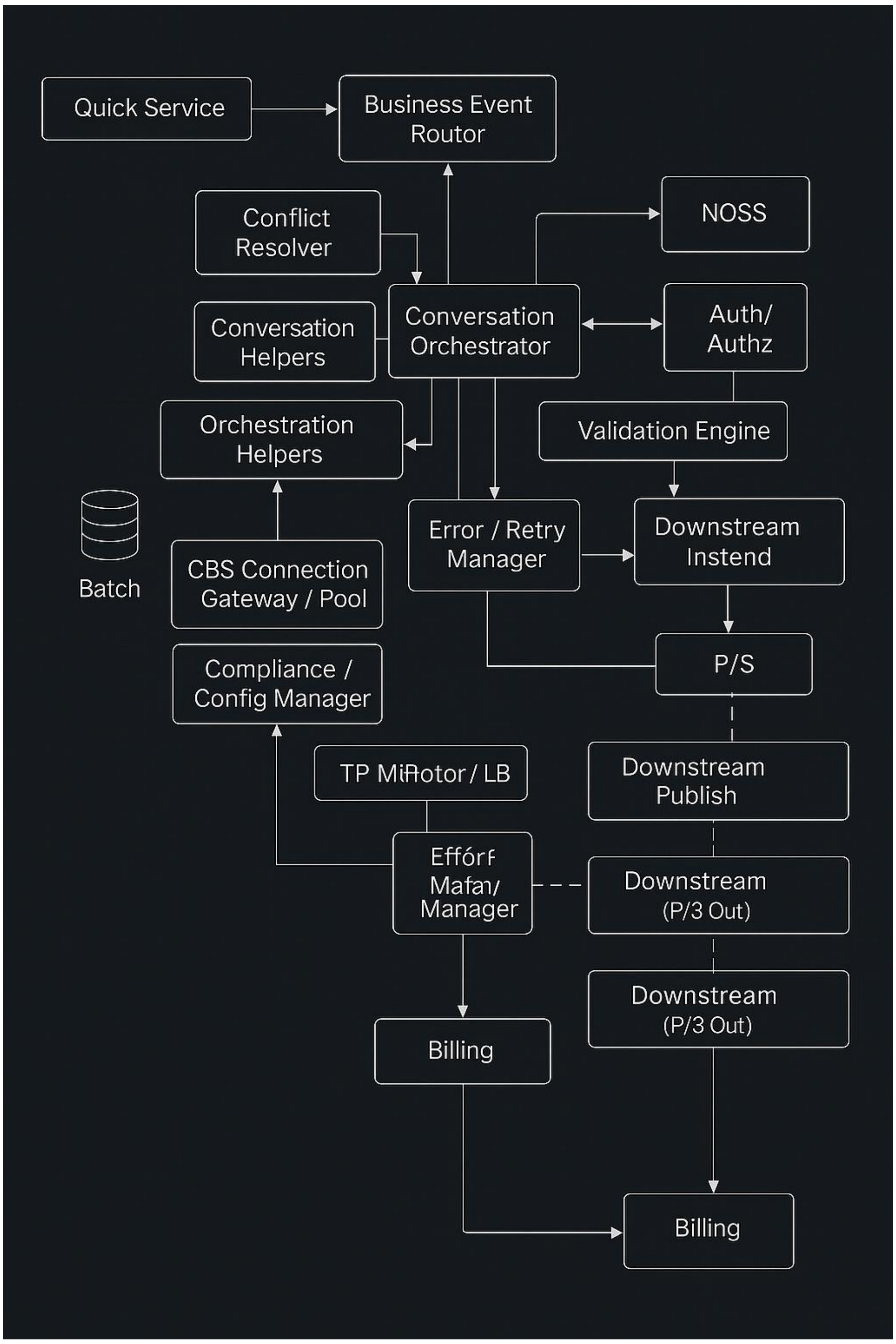
C4 Case Study

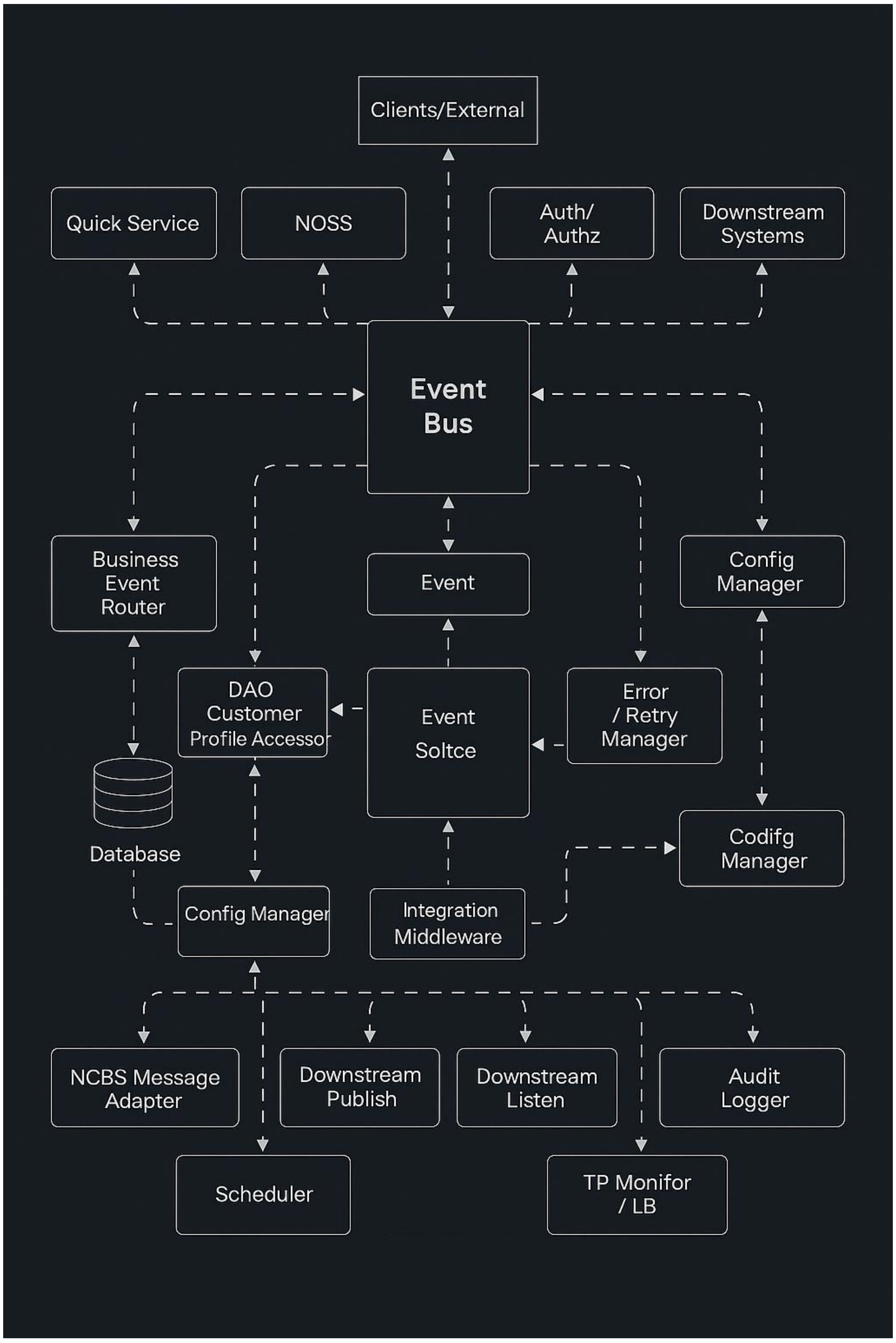
Q1.

I am going with Event-based Style and Microservices style.

Microservices Style:



Event-Based Style:



Q2.

Microservices: rationale

C4 must integrate with NOSS, publish through P/S to multiple downstream systems, and hit online and batch corporate databases while supporting retries, audits, and validation. Those are independent concerns with different change and scaling profiles. Microservices lets me isolate them into small services behind an API gateway and a registry. Conversation orchestration handles long running flows while services like validation, product catalog, availability, and resource locker scale and deploy on their own. This reduces blast radius, matches the enterprise context, and keeps the external L1 interfaces intact.

Microservices: pros and cons, weighed

Pros: independent deployment and scaling for hot spots like catalog or validation, resilience through failure isolation and per service retries.

Cons: higher operational overhead from distributed deployments, harder end to end data consistency.

Given C4's heterogenous integrations and the need to evolve business rules without stopping the world, the pros dominate. I mitigate the cons with a thin platform layer: API gateway, registry, metrics, circuit breakers, idempotency, and compensation via a saga orchestrator.

Event based: rationale

C4 must react to external and internal events such as product availability, downstream acknowledgments, and batch database updates. An event bus removes tight coupling and lets producers and consumers evolve independently. It also matches the existing P/S connector to downstream systems and supports both push and pull variants. Inside the expanded C4, components publish facts and subscribe to what they need while the bus handles fan out and loose coupling.

Event based: pros and cons, weighed

Pros: very low coupling and easy extensibility since new consumers can be added without touching producers, good scalability under bursty workloads through buffering on the bus.

Cons: delivery and ordering are not guaranteed without extra work, debugging and observability are harder because control flow is implicit.

For C4 the benefits are strong since many integrations are asynchronous by nature. I address the risks with durable topics, clear event schemas and versions, idempotent consumers, tracing with correlation IDs, and dead letter handling.

Q3.

When comparing the two architectures, the event-based style addresses the requirement of publishing to multiple downstream systems in a superior way. The case study specifies that C4 must interact with corporate databases, downstream systems, and batch processes while supporting asynchronous acknowledgments. An event bus makes this communication loose and scalable. Producers only emit events and are not concerned with who consumes them, which means downstream systems can be added or removed without affecting the core design. This also handles bursty loads effectively because events can be buffered and delivered as resources allow. Microservices, by contrast, would need explicit service calls for each downstream integration, creating tighter coupling and more maintenance overhead as new consumers are introduced.

On the other hand, microservices address the requirement of complex business orchestration in a stronger way. The case study highlights the need for validation, catalogue lookup, availability checks, resource locking, retries, and audits before committing an order. Microservices allow each of these steps to be implemented as an independent service with well-defined APIs, giving precise control over sequencing, compensation, and recovery. An orchestrator service can coordinate the flow, ensuring consistency across steps. Event based architectures could achieve the same through sagas and distributed events, but tracing, debugging, and enforcing strict ordering would be much more difficult. For this reason, microservices are the superior choice when the requirement is reliable transaction management and clear accountability across business processes.

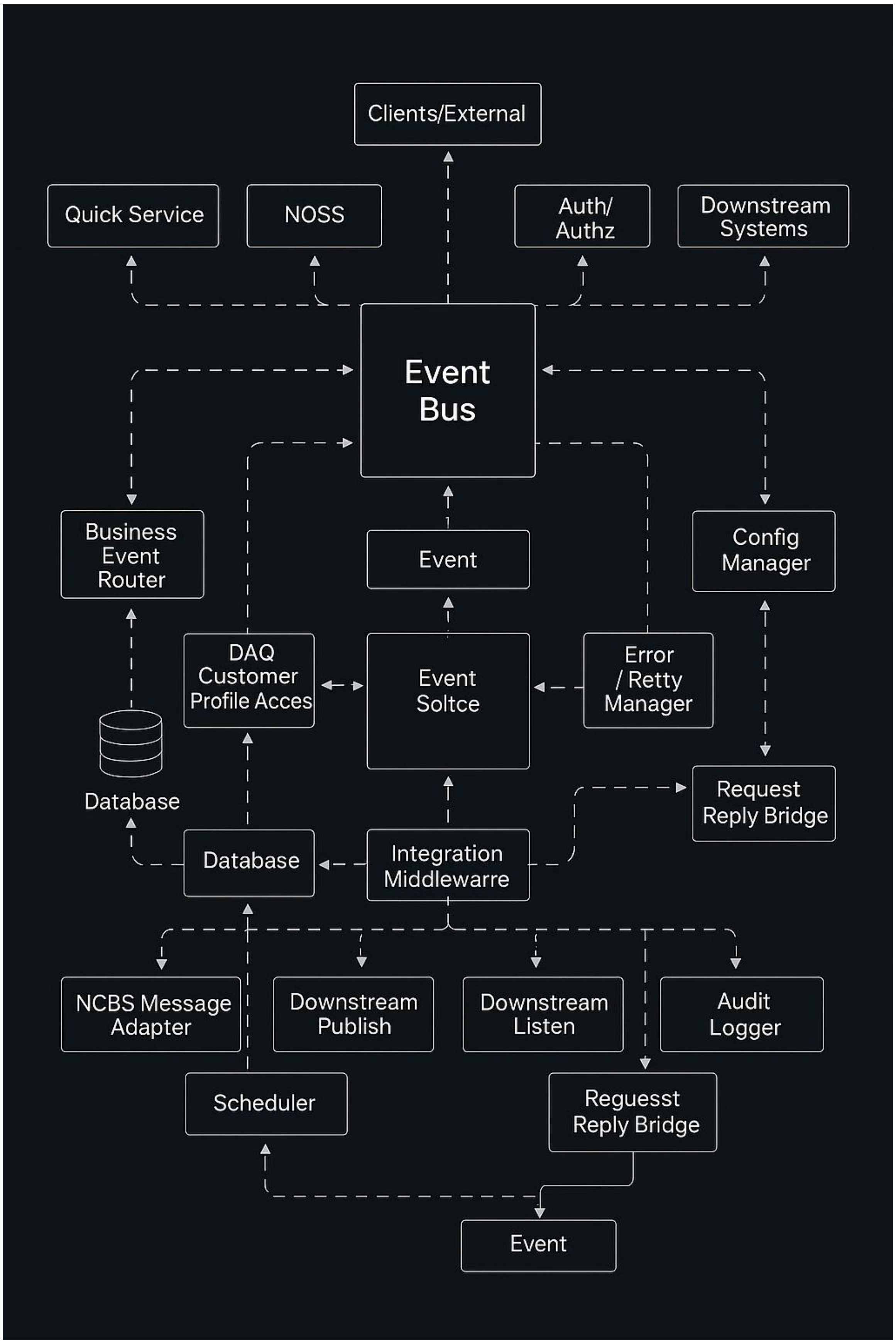
Q4.

The event-based architecture does not fully support three of C4's important requirements. First, it struggles with transactional consistency, particularly when resource locks and validations must be coordinated in real time. Because events are processed asynchronously, ordering is not guaranteed, which makes it difficult to ensure that dependent operations occur in the correct sequence. Second, the style weakens auditability and traceability since decoupled event flows make it hard to reconstruct a clear, sequential record of actions for compliance and reporting. Third, integration with systems like corporate databases that expect synchronous request reply behaviour becomes complex, as the event bus Favors eventual consistency and adds latency when bridging to synchronous APIs.

These shortcomings highlight why event-based design, while powerful for scalability and loose coupling, leaves some critical C4 requirements under served. The architecture naturally supports asynchronous flows and downstream fan out, but it imposes extra work to achieve strong consistency, compliance level logging, and tight integration with synchronous enterprise systems. In practice, these gaps would need to be addressed with compensating patterns such as sagas for transactions, durable event stores with correlation IDs for traceability, and service gateways for database access. Without these additions, the event-based approach cannot fully satisfy C4's most stringent operational needs.

Q5.

I modify the Event based architecture by adding a request reply bridge for synchronous clients, a process manager for sagas, correlation and idempotency services, and durable infrastructure around the bus that includes an event store, a dead letter queue, and an outbox publisher. All communication among C4 components still goes through the event bus so the style is preserved. The bridge converts client requests into commands on the bus and waits for correlated replies without introducing direct component calls. The process manager coordinates validation, availability, and resource locking with compensations, while the event store and correlation service restore sessions after interruptions. The outbox publisher and a database gateway handle integration with downstream systems and corporate databases without bypassing the bus inside C4. This change is appropriate because it addresses transactional consistency, traceability, and session continuity without violating the event-based constraint that components communicate through events.


Clients/External
Quick Service
NOSS
Auth/ Authz
Downstream Systems
Event Bus
Business Event Router
Event
Config Manager
DAQ Customer Profile Acces
Event Soltce
Error / Retty Manager
Database
Request Reply Bridge
Database
Integration Middlewarre
NCBS Message Adapter
Downstream Publish
Downstream Listen
Audit Logger
Scheduler
Reguesst Reply Bridge
Event

Q6.

In the microservices architecture, I would add a dedicated logging microservice that records interactions between C4 and downstream systems. The API gateway or service mesh can monitor the number of active users, and only when the count falls below 100 would it forward interaction details to the logging microservice. This change fits within the microservices style since logging is isolated as its own service with a clear API, preserving loose coupling. The conditional enablement is handled by the orchestration layer, which keeps the rest of the system untouched. The main cost is the added operational logic in monitoring and routing, but the style itself supports this extension naturally.

In the event-based architecture, I would add a logging subscriber that listens to events published by C4 when interactions occur with downstream systems. The event bus or its manager would check the active user count, and only when it is below 100 would it forward the events to the logging subscriber. This also respects the style since the new component simply consumes events without introducing direct coupling. Compared to microservices, the event-based version is easier to change because logging can be added as a new subscriber without altering existing producers. The system can evolve by attaching or detaching consumers, which makes logging integration simpler and less intrusive.