Chapter 1: Event Streaming Fundamentals

It is amazing how the software engineering landscape has transformed over the last decade. Not long ago, applications were largely monolithic in nature, internally-layered, typically hosted within application servers and backed by 'big iron' relational databases with hundreds or thousands of interrelated tables. Distributed applications were the 'gold standard' by those measures — coarse-grained deployable units scattered among a static cluster of application servers, hosted on a fleet of virtual machines and communicating over SOAP-based APIs or message queues. Containerisation, cloud computing, elasticity, ephemeral computing, functions-as-a-service, immutable infrastructure — all niche concepts that were just starting to surface, making minor, barely perceptible ripples in an architectural institution that was otherwise well-set in its ways.

That was then. Today, these concepts are profoundly commonplace. Engineers are often heard interleaving several such terms in the same sentence; it would seem that the engineering community had miraculously stumbled upon an elixir that has all but cured us of our prior burdens — at least when it comes to developer velocity, time-to-market, system availability, scalability, and just about every other material concern that had kept the engineering manager of yore awake at night. Today, we have microservices in the cloud. Problem solved. Next question.

Except no such event *actually* occurred. We did not discover a solution to the problem; we merely shifted the problem. Aspects of software development that used to be straightforward in the 'old world', such as debugging, profiling, performance management, and state consistency — are now an order of magnitude more complex. On top of this, a microservices architecture brings its own unique woes. Services are more fluid and elastic, and tracking of their instances, their versions and dependencies is a Herculean challenge that balloons in complexity as the component landscape evolves. To top this off, services will fail in isolation, further exacerbated by unreliable networks, potentially leaving some activities in a state of partial completeness. Given a large enough system, parts of it may be suffering a minor outage at any given point in time, potentially impacting a subset of users, quite often without the operator's awareness.

With so many 'moving parts', how does one stay on top of these challenges? How does one make the engineering process *sustainable*? Or should we just write off the metamorphosis of the recent decade as a failed experiment?

The real challenges of distributed systems

If there is one thing to be learned from the opening gambit, it is that there is no 'silver bullet'. Architectural paradigms are somewhat like design patterns, but broader scoped, more subjective, and far less prescriptive. However fashionable and blogged-about these paradigms might be, they only offer partial solutions to common problems. One must be mindful of the context at all times, and apply the model judiciously. And crucially, one must understand the deficiencies of the proposed approach, being able to reason about the implications of its adoption — both immediate and long-term.

The principal inconvenience of a distributed system is that it shifts the complexity from the innards of a service implementation to the notional fabric that spans across services. Some might say, it lifts the complexity from the *micro* level to the *macro* level. In doing so, it does not reduce the net complexity; on the contrary, it increases the aggregate complexity of the combined solution. An astute engineering leader is well-aware of this. The reason why a distributed architecture is often chosen — assuming it is chosen correctly — is to enable the compartmentalisation of the problem domain. It can, if necessary, be decomposed into smaller chunks and solved in partial isolation, typically by different teams — then progressively integrated into a complete whole. In some cases, this decomposition is deliberate, where teams are organised around the problem. In other, less-than-ideal cases, the breakdown of the problem is a reflection of Conway's Law, conforming to organisational structures. Presumed is the role an architect, or a senior engineering figure that orchestrates the decomposition, assuming the responsibility for ensuring the conceptual integrity and efficacy of the overall solution. Centralised coordination may not always be present — some organisations have opted for a more democratic style, whereby teams act in concert to decompose the problem organically, with little outside influence.

Coupling

Whichever the style of decomposition, the notion of *macro* complexity cannot be escaped. Fundamentally, components must communicate in one manner or another, and therein lies the problem: components are often inadvertently made aware of each other. This is called *coupling* — the degree of interdependence between software components. The lower the coupling, the greater the propensity of the system to evolve to meet new requirements, performance demands, and operational challenges. Conversely, tight coupling shackles the components of the system, increasing their mutual reliance and impeding their evolution.

There are known ways for alleviating the problem of coupling, such as the use of an asynchronous communication style and message-oriented middleware to segregate components. These techniques have been used to varying degrees of success; there are times where message-based communication has created a false economy — collaborating components may still be transitively dependent upon one another in spite of their designers' best efforts to forge opaque conduits between them.

Resilience

It would be rather nice if computers never failed and networks were reliable; as it happens, reality differs. The problem is exacerbated in a distributed context: the likelihood of any one component experiencing an isolated failure increases with the total number of components, which carries negative ramifications if components are interdependent.

Distributed systems typically require a different approach to resilience compared to their centralised counterparts. The quantity and makeup of failure scenarios is often much more daunting in distributed systems. Failures in centralised systems are mostly characterised as *fail-stop* scenarios — where a process fails totally and permanently, or a network partition occurs, which separates the entirety of the system from one or more clients, or the system from its dependencies. At either rate, the failure modes are trivially understood. By contrast, distributed systems introduce the concept of *partial failures*, *intermittent failures*, and, in the more extreme cases, *Byzantine failures*. The latter represents a special class of failures where processes submit incorrect or misleading information to unsuspecting peers.

Consistency

Ensuring state consistency in a distributed system is perhaps the most difficult aspect to get right. One can think of a distributed system as a vast state machine, with some elements of it being updated independently of others. There are varying levels of consistency, and different applications may demand specific forms of consistency to satisfy their requirements. The stronger the consistency level, the more synchronisation is necessary to maintain it. Synchronisation is generally regarded as a difficult problem; it is also expensive — requiring additional resources and impacting the performance of the system.

Cost being held a constant, the greater the requirement for consistency, the less distributed a system will be. There is also a natural counterbalance between consistency and availability, identified by Eric Brewer in 1998. The essence of it is in the following: distributed systems must be tolerant of network partitions, but in achieving this tolerance, they will have to either give up consistency or availability guarantees. Note, this conjecture does not claim that a consistent system cannot simultaneously be highly available, only that it must give up availability if a network partition does occur.

By comparison, centralised systems are not bound by the same laws, as they don't have to contend with network partitions. They can also take advantage of the underlying hardware, such as CPU cache lines and atomic operations, to ensure that individual threads within a process maintain consistency of shared data. When they do fail, they typically fail as a unit — losing any ephemeral state and leaving the persistent state as it was just before failure.

Event-Driven Architecture

Event-Driven Architecture (EDA) is a paradigm promoting the production, detection, consumption of, and reaction to *events*. An event is a significant state in change, that may be of interest within the domain where this state change occurred, or outside of that domain. Interested parties can be notified of an event by having the originating domain publish some canonical depiction of the event to a well-known conduit — a message broker, a ledger, or a shared datastore of some sort. Note, the event itself does not travel — only its notification; however, we often metonymically refer to the notification of the event as the event. (While formally incorrect, it is convenient.)

An event-driven system formally consists of *emitters* (also known as producers and agents), *consumers* (also known as subscribers and sinks), and *channels* (also known as brokers). We also use the term *upstream* — to refer to the elements prior to a given element in the emitter-consumer relation, and *downstream* — to refer to the subsequent elements.

An emitter of an event is not aware of any of the event's downstream consumers. This statement captures the essence of an event-driven architecture. An emitter does not even know whether a consumer exists; every transmission of an event is effectively a 'blind' broadcast. Likewise, consumers react to specific events without the knowledge of the particular emitter that published the event. A consumer need not be the final destination of the event; the event notification may be persisted or transformed by the consumer before being broadcast to the next stage in a notional pipeline. In other words, an event may spawn other events; elements in an event-driven architecture may combine the roles of emitters and consumers, simultaneously acting as both.

Event notifications are immutable. An element cannot modify an event's representation once it has been emitted, not even if it is the originally emitter. At most, it can emit new notifications relating to that event — enriching, refining, or superseding the original notification.

Coupling

Elements within EDA are exceedingly loosely coupled, to the point that they are largely unaware of one another. Emitters and consumers are only coupled to the intermediate channels, as well as to the representations of events — *schemas*. While some coupling invariably remains, in practice, EDA offers the lowest degree of coupling of any practical system. The collaborating components become largely autonomous, standalone systems that operate in their own right — each with their individual set of stakeholders, operational teams, and governance mechanisms.

By way of an example, an e-commerce system might emit events for each product purchase, detailing the time, product type, quantity, the identity of the customer, and so on. Downstream of the emitter, two systems — a business intelligence (BI) platform and an enterprise resource planning (ERP) platform — might react to the sales events and build their own sets of materialised views. (In effect, view-only projections of the emitter's state.) Each of these platforms are completely independent systems with their own stakeholders: the BI system satisfies the business reporting and analytics requirements for the marketing business unit, while the ERP system supports supply chain management and capacity planning — the remit of an entirely different business unit.

To put things into perspective, we shall consider the potential solutions to this problem in the absence of EDA. There are several ways one could have approached the solution; each approach commonly found in the industry to this day:

1. **Build a monolith**. Conceptually, the simplest approach, requiring a system to fulfill all requirements and cater to all stakeholders as an indivisible unit.
2. **Integration**. Allow the systems to invoke one another via some form of an API. Either the e-commerce platform could invoke the BI and ERP platforms at the point of sale, or the BI and ERP platforms could invoke the e-commerce platform APIs just before generating a business

report or supplier request. Some variations of this model use message queues for systems to send commands and queries to one another.

3. **Data decapsulation**. If system integrators were cowboys, this would be their prairie. Data decapsulation (a coined term, if one were to ask) sees systems 'reaching over' into each other's 'backyard', so to speak, to retrieve data directly from the source (for example, from an SQL database) — without asking the owner of the data, and oftentimes without their awareness.
4. **Shared data**. Build separate applications that share the same datastore. Each application is aware of all data, and can both read and modify any data element. Some variations of this scheme use database-level permissions to restrict access to the data based on an application's role, thereby binding the scope of each application.

Once laid out, the drawbacks of each model become apparent. The first approach — the proverbial monolith — suffers from uncontrolled complexity growth. In effect, it has to satisfy *everyone* and *everything*. This also makes it very difficult to change. From a reliability standpoint, it is the equivalent of putting all of one's eggs in one basket — if the monolith were to fail, it will impact all stakeholders simultaneously.

The second approach — integrate everything — is what these days is becoming more commonly known as the 'distributed monolith', especially when it is being discussed in the context of microservices. While the systems (or services, as the case may be) appear to be standalone — they might even be independently sourced and maintained — they are by no means autonomous, as they cannot change freely without impacting their peers.

The third approach — read others' data — is the architectural equivalent of a 'get rich quick scheme' that always ends in tears. It takes the path of least resistance, making it highly alluring. However, the model creates the tightest possible level of coupling, making it very difficult to change the parties down the track. It is also brittle — a minor and seemingly benign change to the internal data representation in one system could have a catastrophic effect on another system.

The final model — the use of a shared datastore — is a more civilised variation of the third approach. While it may be easier to govern, especially with the aid of database-level access control — the negative attributes are largely the same.

Now imagine that the business operates *multiple* disparate e-commerce platforms, located in different geographic regions or selling different sorts of products. And to top it off, the business now needs a separate data warehouse for long-term data collection and analysis. The addition of each new component significantly increases the complexity of the above solutions; in other words, they do not scale. By comparison, EDA scales perfectly linearly. Systems are unaware of one another and react to discrete events — the origin of an event is largely circumstantial. This level of autonomy permits the components to evolve rapidly in isolation, meeting new functional and non-functional requirements as necessary.

Resilience

The autonomy created by the use of EDA ensures that, as a whole, the system is less prone to outage if any of its individual components suffer a catastrophic failure. How is this achieved?

Integrated systems, and generally, any topological arrangement that exhibits a high degree of component coupling is prone to *correlated failure* — whereby the failure of one component can take down an entire system. In a tightly coupled system, components directly rely on one another to jointly achieve some goal. If one of these components fails, then the remaining components that depend on it may also cease to function; at minimum, they will not be able to carry out those operations that depend on the failed component.

In the case of a monolith, the failure assertion is trivial — if a fail-stop scenario occurs, the entire process is affected.

Under EDA, enduring a component failure implies the inability to either emit events or consume them. In the event of emitter failure, consumers may still operate freely, albeit without a facility for reacting to new events. Using our earlier example, if the e-commerce engine fails, none of the downstream processes will be affected — the business can still run analytical queries and attend to resource planning concerns. Conversely, if the ERP system fails, the business will still make sales; however, some products might not be placed on back-order in time, potentially leading to low stock levels. Furthermore, provided the event channel is durable, the e-commerce engine will continue to publish sales events, which will eventually be processed by the ERP system when it is restored. The failure of an event channel can be countered by implementing a local, stateful buffer on the emitter, so that any backlogged events can be published when the channel has been restored. In other words, not only is an event-driven system more resilient by retaining limited operational status during component failure, it is also capable of self-healing when failed components are replaced.

In practice, systems may suffer from soft failures, where components are saturated beyond their capacity to process requests, creating a cascading effect. In networking, this phenomenon is called 'congestive collapse'. In effect, components appear to be online, but are stressed — unable to turn around some fraction of requests within acceptable time frames. In turn, the requesting components — having detected a timeout — retransmit requests, hoping to eventually get a response. This increases pressure on the stressed components, exacerbating the situation. Often, the missed response is merely an indication of receiving the request — in effect, the requester is simply piling on duplicate work.

Under EDA, requesters do not require a confirmation from downstream consumers — a simple acknowledgement from the event channel is sufficient to assume that the event has been stably enqueued and that the consumer(s) will get to it at some future point in time.

Consistency

EDA ameliorates the problem of distributed consistency by attributing explicit mastership to state, such that any stateful element can only be manipulated by at most one system — its designated owner. This is also referred to as the originating domain of the event. Other domains may only react to the event; for example, they may reduce the event stream to a local projection of the emitter's state.

Under this model, consistency within the originating domain is trivially maintained by enforcing the single writer principle. External to the domain, the events can be replayed in the exact order they were observed on the emitter, creating *sequential consistency* — a model of consistency where updates do not have to be seen instantaneously, but must be presented in the same order to all observers, which is also the order they were observed on the emitter. Alternatively, events may be emitted in *causal order*, categorising them into multiple related sequences, where events within any sequence are related amongst themselves, but unrelated to events in another sequence. This is a slight relaxation of sequential consistency to allow for safe parallelism, and is sufficient in the overwhelming majority of use cases.

Applicability

For all its outstanding benefits, EDA is not a panacea and cannot supplant integrated or monolithic systems in all cases. For instances, EDA is not well-suited to synchronous interactions, as mutual or unilateral awareness among collaborating parties runs contrary to the grain of EDA and negates most of its benefits.

EDA is not a general-purpose architectural paradigm. It is designed to be used in conjunction with other paradigms and design patterns, such as synchronous request-response style messaging, to solve more general problems. In the areas where it can be applied, it ordinarily leads to significant improvements in the system's non-functional characteristics. Therefore, one should seek to maximise opportunities for event-driven compositions, refactoring the architecture to that extent.

What is event streaming?

Finally, we arrive at the central question: *What is event streaming?* And frankly, there is little left to explain. There is but one shortfall in the earlier narrative: EDA is an architectural paradigm — it does not prescribe the particular semantics of the event interchange. Events could be broadcast among parties using different mechanisms, all potentially satisfying the basic tenets of EDA.

Event streaming is a mechanism that can be used to realise the *event channel* element in EDA. It is primarily concerned with the following aspects of event propagation:

- Interface between the emitter and the channel, and the consumer and the channel;
- Cardinality of the emitter and consumer elements that interact with a common channel;
- Delivery semantics;
- Enabling parallelism in the handling of event notifications;
- Persistence, durability, and retention of event records; and
- Ordering of events and associated consistency models.

The focal point of event streaming is, unsurprisingly, an *event stream*. At minimum, *an event stream is a durable, totally-ordered, unbounded sequence of immutable event records, delivered at least once to its subscriber(s).* An *event streaming platform* is a concrete technology that implements the event streaming model, addressing the points enumerated above. It interfaces with emitter and consumer ecosystems, hosts event streams, and may provide additional functionality beyond the essential set of event streaming capabilities. For example, an event streaming platform may offer end-to-end compression and encryption of event records, which is not essential in the construction of event-driven systems, but is convenient nonetheless.

It is worth noting that event streaming is not required to implement the event channel element of EDA. Other transports, such as message queues, may be used to fulfill similar objectives. In fact, there is nothing to say that EDA is exclusive to distributed systems; the earliest forms of EDA were realised within the confines of a single process, using purely in-memory data structures. It may seem banal in comparison, but even UI frameworks of the bygone era, such as Java Swing, draw on the foundations of EDA, as do their more contemporary counterparts, such as React.

When operating in the context of a distributed system, the primary reason for choosing event streaming over the competing alternatives is that the former was designed specifically for use in EDA, and its various implementations — event streaming platforms — offer a host of capabilities that streamline their adoption in EDA. A well-designed event streaming platform provides direct correspondence with native EDA concepts. For example, it takes care of event immutability, record ordering, and supports multiple independent consumers — concepts that might not necessarily be endemic to alternate solutions, such as message queues.

This chapter has furnished an overview of the challenges of engineering distributed systems, contrasted with the building of monolithic business applications. The numerous drawbacks of distributed systems increase their cost and complicate their upkeep. Generally speaking, *the components of a complex system are distributed out of necessity* — namely, the requirement to scale in both the performance plane and in the engineering capacity to deliver change.

We looked at how the state of the art has progressed since the mass adoption of the principles of distributed computing in mainstream software engineering. Specifically, we explored *Event-Driven Architecture* as a highly effective paradigm for reducing coupling, bolstering resilience, and avoiding the complexities of maintaining a globally consistent state.

Finally, we touched upon *event streaming*, which is a rendition of the *event channel* element of EDA. We also learned why event streaming is the preferred approach for persisting and transporting event notifications. In no uncertain terms, event streaming is the most straightforward path for the construction of event-driven systems.