D I Z Z Y

Foundations for Interstitial Infrastructure

Conrad Mearns

> "The required techniques of effective reasoning are pretty formal, but as long as programming is done by people that don't master them, the software crisis will remain with us and will be considered an incurable disease."
>
> — (Dijkstra 2000)

Abstract

Software projects fail in predictable ways. Infrastructure decisions made under pressure harden into permanent constraints. Domain logic becomes inseparable from the languages that encode it. Data and services become inseparable from the databases that serve. The vocabulary used to describe a system drifts from the code that implements with every commit, requiring continuous and careful maintenance. DIZZY is a philosophy for building event-driven software systems that keeps domain logic, data contracts and infrastructure permanently and deliberately separate. It draws on established practices - Event Driven Architectures, Command Query Responsibility Separation, Event Sourcing, and Domain-Driven Design - and composes them into a coherent discipline: define the domain in a language-agnostic schema, generate typed contracts for data and code, and let infrastructure decisions follow rather than lead. **Interstitial Infrastructure** as a practice is then defined as the technique for deriving such infrastructures. This paper is written for software practitioners and technical-decision-makers who have felt these problems firsthand. It makes the case for why the separations DIZZY enforces matter, introduces a system of eight program components that illustrates the philosophy, and describes a software generation pipeline that closes the gap between domain discovery and working, scalable software.

Last Updated: 2026-05-25

Table of Contents

1. The Burden of Software Architecture is…

1.1. Building the Wrong Things

1.2. The Irreversibility of Structural Decisions

1.3. Scaling on Budget and on Time

2. Software Architectures Should…

…deliberately shape the communication structures of the organizations that build them.

> [I]t is often worth investigating whether restructuring your organization or team would prevent the new application from displaying all the same structural dysfunctions as the original
>
> — (LeRoy, Simons 2010)

The Inverse Conway Maneuver says: if your system is going to mirror your organization anyway, design it to mirror the organization you *want*. DIZZY applies this as an architecture constraint. Its rigid component boundaries are deliberate seams that define where teams own work independently and where they must coordinate.

2.1. Define Processes for Stakeholder Engagement

2.2. Facilitate an Understanding of What the Software Does

2.3. Be Predictable to Build, Task, and Scale

3. DIZZY Solves this by...

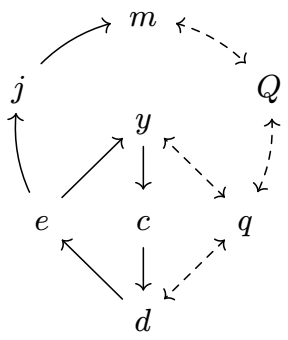
...through it's philosophy

3.1. High to Low LoD w/ only Business concerns

3.2. Transforming Viable Software to Scalable Software

3.3. Predictability, Scaling, Task Partitioning

4. Implementation



DIZZY is Data Oriented and Functional. We explicitly outline each data contract component and each function component separately.

The diagram to the left shows the general data flow between each DIZZY data and function component. Solid arrows show message passing as messages passed via queues. Dashed arrows show call-and-response communications, or subscription-based streams.

The implementation of DIZZY is oriented around message passing. While this creates many new issues like Race Conditions (discussed in Section 6.3) — the primary hypothesis of DIZZY is that these problems are solvable algorithmically.

Each component represents a unit of computation that requires careful considerations around its intended semantic constraints. DIZZY is an Event Driven paradigm, that assumes Event Sourcing is used for a source of truth. This means explicit Command Query Responsibility Separation (CQRS)

Symbol	Component	Role
c	Command	Represents an intent - what a user or policy wants to happen
d	Procedure	Handles a Command and may emit any Events
e	Event	Immutable record of facts as the source of truth, intended to be stored with Event Sourcing
y	Policy	Reacts to an Event and may emit any Commands
m	Model	Typically, a database schema. Used not as a source of truth - but an engine for optimizing queries.
j	Projection	Triggered for Events and updates Models. The mutation ORM layer.
q	Query	Typed contract for a call and its response. Represented as an Input and Output tuple
Q	Querier	Executes a Input Query against a Model

DIZZY is therefore comprised of two dataflow loops.

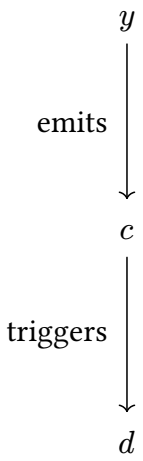
Firstly, a reactivity loop; Where Commands trigger Procedures, which emit Events that trigger Policies, that emit Commands. Secondly, a data retrieval loop; Where Event are projected to Models (databases), which can be queried by Procedures and Policies. The reactivity loop enables processing, algorithms, and business logic to react and change to new information. The retrieval loop enables database decoupling while providing efficient value lookups.

As an example, suppose we processing financial transactions. Each Events record debits and credits as the source of truth. Transactions are initiated by users and the data is represented by a Command. The Transaction Command triggers a procedure, which may need to Query for the

normalized account balance to know whether to succeed or fail. Thus, each debit and credit Event must also trigger a projection to keep this normalized account balance up-to-date in a Model.

4.1. Data and Function Components

4.1.1. Commands (c) that Represent Intent



Commands represent expressions of intent, what users *want* to happen. They are ephemeral - not records of fact, and thus may be fire-and-forget and discarded. This distinction matters: intent can be rejected, retried, or rerouted.

Facts cannot.

Because Commands are ephemeral and may be discarded without consequence, the are useful for isolating the transmission of data must abide to different legal and business policies - such as personally identifiable information.

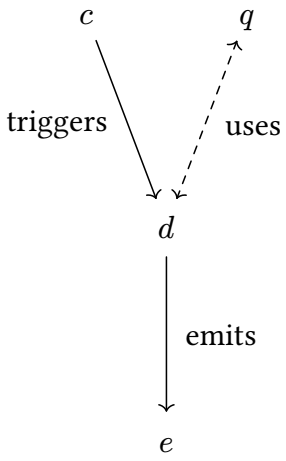
Whether this information is recorded as an immutable fact is up to the design of Procedures and Events.

A policy may emit any number of Commands, of any Command sub-types. Therefore policies and Commands are one-to-many; several policies may emit the same Command types.

A Command may trigger many Procedures. This relation is typically best assigned at the Procedure definition.

Commands express intent, and can be carefully designed to handle cases where lossy systems where retries are standard practice. See Durable Execution

4.1.2. Procedures (d) that Perform the Critical Work

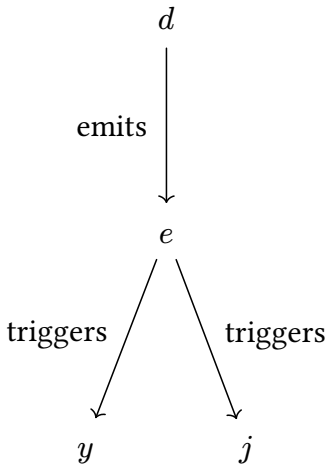


Procedures handle Commands and may emit Events. They are the site of critical work in a DIZZY system — the place where intent meets consequence.

Procedures are not required to be pure functions. They may query Models to inform their decisions. But any external system a Procedure touches must either be modelled as another DIZZY component, or its effects must be recorded in Events. This constraint is what makes Procedures portable across languages and deployable in any topology: their behavior is fully described by what they receive, what they query, and what they emit.

When a Procedure performs IO or causes effects in the world, it should emit Events to record the attempt, progress, and result. An effect that is not recorded in an Event is invisible to the rest of the system — and invisible effects undermine the audit guarantee that Events provide.

4.1.3. Events (e) that are the Source of Truth



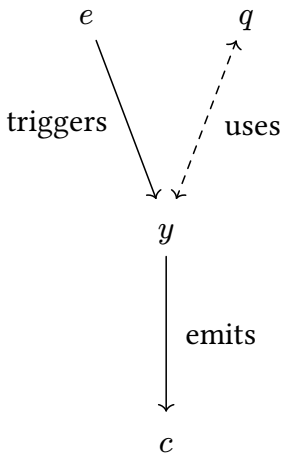
Events are the single source of truth. Everything else — every model, every view, every report — must be derived from events. Events are immutable records of what *happened*: once written, they cannot be altered or deleted.

Because events capture what happened independently of any particular database schema, the same event stream can power multiple databases with completely different schemas — each optimized for different queries. Different teams or services can maintain their own models from the shared event stream; coordination happens through the event contract, not through shared database access.

This also makes migrations tractable. Rather than transforming a live database in place, a new Model can be built alongside the old one from the same event history, and cut over when ready. If the new schema turns out to be wrong, the events remain — and the next attempt costs only a new Projection.

The event stream is the canonical representation. Every database is just a cached, queryable projection of that truth — disposable and rebuildable by design.

4.1.4. Policies (y) that React and Initiate

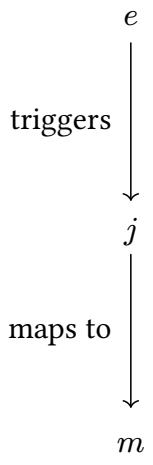


Policies are the reactive logic of a DIZZY system. Where Procedures respond to intent, Policies respond to fact: they are triggered by Events that have already occurred, and they may emit Commands in response. A Policy asks: *given that this happened, what should happen next?*

This distinction is not incidental. Because a Policy reacts to an Event — an immutable, already-happened fact — it is inherently decoupled from the Procedure that produced it. The Policy does not know or care how the Event came to exist. It knows only what happened, and responds accordingly.

Policies may also query Models to inform their decisions, allowing the system to react differently depending on current state. A Policy that always emits the same Command is a simple rule. A Policy that queries a Model first is a conditional one — but the conditionality lives in the Policy, not scattered through unrelated code.

4.1.5. Projections (j) that Map Events to Models

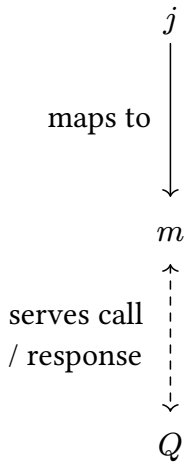


Events are the authoritative record of everything that has happened, but they are not optimized for retrieval. A Projection solves this: it listens for specific Events and updates one or more Models in response, translating the language of facts into the language of efficient queries.

This approach eliminates the need for Aggregates — the construct traditional Domain-Driven Design uses to bridge event-driven architecture to object-oriented state. Because Models in DIZZY are treated as ephemeral and rebuildable from Events, there is no need for an object that holds and manages state on behalf of a set of related entities.

The practical consequence is significant: if a Model turns out to be wrong for its purpose, it can be replaced without touching the event history. Write a new Projection, replay the Events, and a new Model emerges from the same source of truth.

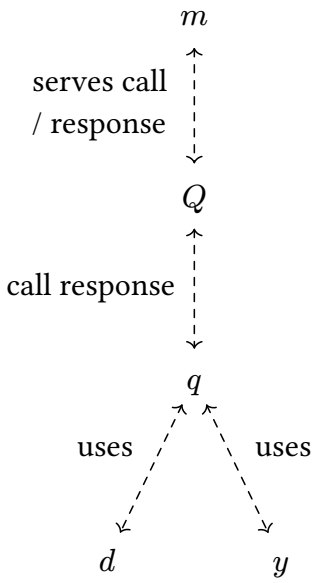
4.1.6. Models (m) that Serve Data for Queries



A Model is a schema for queryable state, backed by a database chosen for the specific access patterns that Model must serve. Models are derived from Events via Projections, and are rebuildable on demand.

Models exist to make queries fast, not to be the source of truth. You can have as many Models as you have distinct questions to answer — each backed by whatever database technology best suits those questions. A time-series store for metrics, a graph database for relationships, a search index for full-text: all derived from the same Event stream.

4.1.7. Queries (q) and Queriers (Q) for efficient computation



Querying can often be the most confused aspect of software engineering, because the retrieval of information tends to occur at the confluence of database requirements, data shape requirements, and downstream alignment.

DIZZY systems demand the distinction between database drivers, data contracts, and the program that 'executes' the query. We call the data contracts Queries, which are typically represented as a tuple of Input and Output, or Call and Response. The process that uses program code to execute the query is what we call the Querier.

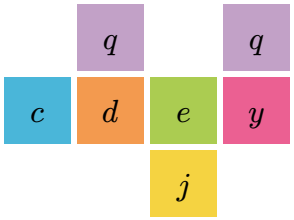
4.2. Architecting Generic Interstitial Infrastructures

4.2.1.

5. How to Structure Software Development Workflows with DIZZY

DIZZY is a complete philosophy for software development; it covers the flow from ideation, vibe-checking, workshopping, integration, to production deployment. This whitepaper will cover the workshopping method, and less of vibe-checking. From there, it will show the tooling to illustrate what steps can be automated algorithmically and which need intervention.

5.1. Workshopping



DIZZY uses a workshop to bring business stakeholders, customers, scientists, and other domain experts together to find consensus around software processes. Computation and Systems experts should serve only to teach, ask questions, and suggest symbolic solutions to process gaps, NOT to define or suggest what would otherwise be the process flow.

The workshop is largely based on Event Storming (Brandolini 2021) by Alberto Brandolini and uses many of the same techniques. Attendees write on sticky notes and place them on a wall next to other notes. Note colors are semantic symbols. Notes convey and narrate a system in the language of facts and intentions. Unlike Event Storming, DIZZY notes represent *real* system components.

Events ( ) pin the most important facets of the process, and are recorded in past tense: "order placed," "payment declined," "shipment dispatched." These records are facts by which we represent all state changes.

Commands ( ) hook process automation and users into the ecosystem by capturing an intent. They are written imperatively: "place order", "submit payment." These Commands may represent a button, API call, a form, or any other method of supplying new instruction to a software system.

Procedures ( ), Policies ( ), Projections ( ), and Queries ( ), replace the "Aggregate", "View", and "Process/Policy" systems of the DDD-inspired Event Storming model.

After the workshop, the DIZZY workflow guides the progressive development of each component until a software artifact is produced. This allows stakeholders to build the software visually and without confounding details that aren't important to those stakeholders.

No database is chosen. No framework is selected. The output is a shared vocabulary in regards to the domain at hand.

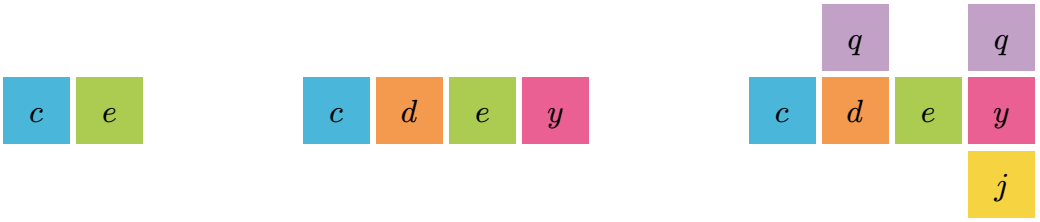
That vocabulary maps directly onto DIZZY's component schema. A workshop session does not merely produce a pile of forgettable photographs; it produces the skeleton of a *feature definition* which the rest of DIZZY transforms into functioning, scalable software.

This directness is intentional. DIZZY's eight components exist precisely so that the output of domain discovery can be written down without loss of meaning. A feature defined in DIZZY's vocabulary is still readable by the domain expert who participated in the session. The command names, event names, and policy reactions are their words - not a translation of them.

This couples the language of the Computationalists and the Domain Experts - allowing both to have a concrete and shared artifact that communicate bidirectionally between abstract graphs and charts - and deployed and running systems.

DIZZY borrows the collaborative spirit of Brandolini's original technique, but adapts the semantic and pragmatic functional paradigm. Aggregates, read models, and the object-oriented constructs of traditional Domain-Driven Design are absent. Every DIZZY component is either a data contract or a stateless process - another distinction that makes components independently deployable, testable, and replaceable in ways the original framing does not require.

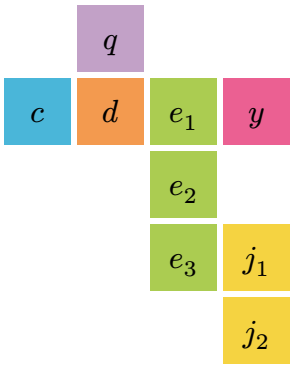
Like Event Storming though, the practice is intended to be simple and progressive. Start by pinning Commands and Events. This naturally exposes what procedures and policies may be required. In turn, this can expose new Commands and Events that my be required for consistency, which prompts more Procedures and Policies - etc.



When it is becomes time to define the gist of how Procedures and Policies function - this is where Projections and Queries come in.

Upon these notes, write the name of Model (representative of a database schema or ontology), and the name of the projection or query if it helps for disambiguation.

For instance, after an "Order Placed" Event, we may need to project the result to two Models - one for "Shipping", and another for "Inventory". Naming these projections further is optional at this phase, but is useful later when the projections require disambiguation.



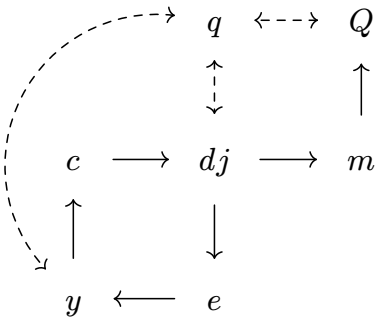
An exact semantics for organizing notes is not needed. If the workshop gets crowded - make copies of notes and spread things out until things make sense.

5.2. Vibechecks - Studying Vibes and Experiments

DIZZY is hard to emulate with simple scripts, because a simple script is usually one that follows the UNIX Philosophy: file in, file out, do one job and do it well. We can fit the DIZZY components to fit the guise of UNIX by composing Procedures and Projections.

dj scripts sacrifice rigidity of the DIZZY core model to serve as an exercise of Intent and Event Extraction.

More on this: https://github.com/ConradMearns/without-objective/tree/main/Structured-Log-Pipes



Here, Q typically reflects a FileSystem. q represents read operations, while j represents writes. Policies (y) aren't explicitly required, but still encode some level of reactivity to events that have taken place.

The goal is *not* to write software like this. Rather, it's to identify that the UNIX philosophy was indeed the standard pattern for writing software, an explicit Structure Log Piping system expresses better flexibility in building small scripts, which can be more easily translated into scalable architectures later.

Using this abstraction, we can now peer into the black box which may be our script - and identify when and where changes to Models (file outputs, database calls) are made.

We can identify IO reads and assign them names as Queries, and we can begin to imagine what Events must exist in order to become DIZZY compatible.

In practice, this can be fairly simple. Often, it is enough to transform the base logging system into one which traces Side-Effects as DEBUG logs. When those logs are structured, and consistent, then we can begin to use the logs themselves as a sort of pseudo Change Data Capture / "Event Store" for projections.

This exercise allows for legacy software to be transformed to become DIZZY compatible.

5.3. Automation

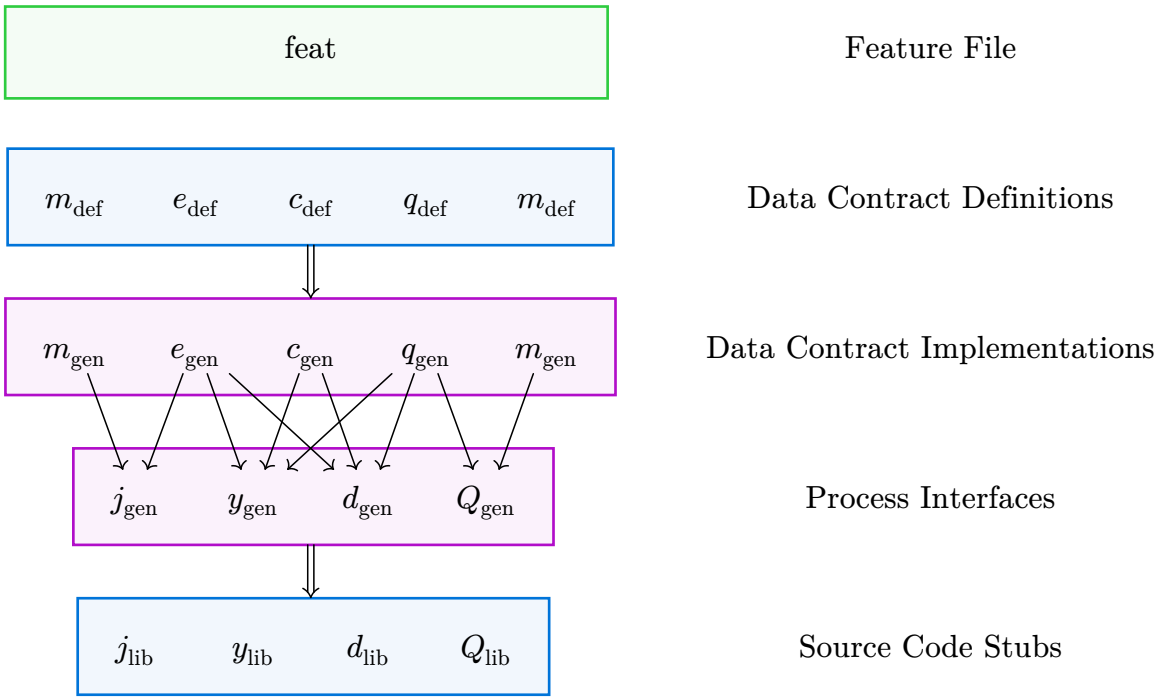


Figure 12: The DIZZY generation pipeline. A Feature File (green) is authored during domain discovery. Data Contract Definitions (blue) are then populated with field-level detail. The generator compiles these into typed Data Contract Implementations and Process Interfaces (purple), which are packaged as runtime-specific library artifacts (blue) that engineers implement against.

The outcome of workshopping is a Feature File. The Feature File is a structured declaration of every component the system requires: which Commands exist, which Procedures handle them, which Events are then produced, which Policies react, which Projections maintain which Models, and which Queries serve those Procedures and Policies.

The Feature File is a map. It documents the high level structures which are then resolved into LinkML schemas and language-specific interfaces.

A developer, a technical lead, or the DIZZY automation tool can look at a Feature File and list exactly what needs to be built. Each Procedure and Policy is a bounded unit of work with explicitly declared inputs, outputs, and data dependencies.

The generation pipeline then takes this map and produces the scaffolding that engineers implement against.

5.3.1. Building Data Definition Schemas

The Feature File names components but does not describe the shape of the data they carry. Building data definitions is the step that gives those names their structure. What fields does a "receipt ingested" event carry? What is the schema of the "receipts" model? These questions are answered by authoring definitions in LinkML - a language-agnostic schema language that represents domain objects in YAML, independent of any particular programming language, framework, or runtime. The Feature File is used to generate LinkML stubs for every Data Definition that is required, details can be filed in after.

5.3.2. Building Program Processes

After Data Definitions are built, and the component wiring declared in the Feature File, typed interfaces are generated for every process. The interface for a Procedure declares exactly which Command it receives, which Queries it is permitted to call, and which Events it may emit. The interface for a Policy declares the Event that triggers it and the Commands it is allowed to dispatch.

At this stage, the The Feature File becomes a compile-time constraints for the architecture. An implementation that exceeds its declared scope is a type error, and can be caught before deployment rather than after.

This generation step makes an explicit claim: the hard architectural decisions have already been made. By the time an engineer receives a generated interface, the question of what each component does - and what it is forbidden from doing - has been resolved at the domain level. What remains is writing the Interstitial Infrastructure to tie the component flows together. The interface enforces the boundary between domain thinking and implementation.

5.3.3. Packaging and Using Libraries

The final step packages the generated interfaces and their compiled data types into importable library artifacts. A Procedure in Python receives a Python library. A Policy in Rust receives a Rust library. A Querier in TypeScript receives a TypeScript library. Each library contains the same domain contracts, as generated by LinkML.

This is the handoff between the Domain and the Deployment.

Engineers implement against the library. The library does not change unless the domain definitions change. Infrastructure decisions, such as which database backs a Model, which message queue carries a Command - are made after the library is built, and they are made at a different layer of the architecture. Domain logic never knows where its inputs came from or where its outputs go.

6. Open Problems

DIZZY does not claim to have solved every hard problem in software architecture. It claims to have made those problems *visible* — to surface them at the domain level where they can be reasoned about, rather than bury them in infrastructure where they are discovered only after deployment.

The following are problems that DIZZY identifies clearly and for which DIZZY's vocabulary provides a precise language — but for which the solutions are not yet fully specified. They are named here because honest acknowledgment of what is hard is more useful than silence.

6.1. Reactive User Interfaces

A User Interface component is not a first-class DIZZY component. It is an abstraction built on top of the Query and Command vocabulary: a pairing of at most one Query and a list of zero or more Commands.

The Query side answers: *what does this view show?* The Command side answers: *what can the user do from here?*

Three configurations cover most practical cases. A form or button that submits without first reading state carries no Query and one or more Commands — the view does not need to know anything about current system state to do its job. A read-only dashboard carries a single Query and no Commands — it shows state and initiates nothing. A fully interactive view carries a Query and two or more Commands — the Query result is context for the available actions (a stock ticker paired with buy and sell Commands, for instance).

6.1.1. Cache Invalidation

The sharp problem for all three configurations is the same: the view eventually goes stale.

In a DIZZY system, state flows in one direction: Events are written, Projections update Models, Queriers answer Queries from those models. When an Event fires and a Projection writes new model state, any active Query whose result reads from that Model may no longer reflect reality.

The challenge is determining *which* Queries are invalidated by *which* Events, and propagating a refresh signal to the right views without broadcasting to all of them.

The Feature File already declares the edges of this graph: each Projection maps specific Event types to specific Models, and each Query targets specific Models. The invalidation relationship from Event to Query is therefore derivable from the same component wiring that defines the rest of the system.

6.1.2. Predicate Pushdown

Knowing *that* a Query might be stale is not sufficient. A user viewing their own order should not receive a refresh notification every time any order in the system changes.

The Query Input already carries the predicate — it is the set of constraints that scoped the original result. A notification system that passes the relevant fields of the triggering Event alongside the invalidation signal allows each UI component to evaluate locally whether its cached result is affected. The broker routes to subscribers of the affected Model; each subscriber decides for itself whether its particular Query is invalidated.

6.2. Authorization and Access Control

DIZZY's component boundaries create a natural surface for authorization, but do not yet specify how it is enforced.

Commands express intent from an actor. Events record facts that have occurred. Queries request information from a Model. Each is a point at which the system can ask: *is this actor permitted to do this?*

The problem is that authorization logic tends to bleed. In conventional systems, access checks accumulate inside business logic — scattered across procedures, database queries, and API handlers — rather than being enforced at a consistent boundary. DIZZY's strict separation makes the problem legible: an authorization rule is a predicate on a Command, an Event, or a Query, and belongs at the component boundary rather than woven through the implementation.

What DIZZY does not yet specify is where those predicates live, how they compose across nested components, and how they interact with audit requirements already carried by the Event stream.

6.3. Race Conditions

Projections before policies? Policies before projections?

Linked deeply with another open problem: Events, Database Normalization, Modeling and Single Source of Truth.

6.4. Modeling Source of Truth

How do you know an event is modelled correctly and fully?

Possibly the same way you determine if a relational database is modelled “correctly”.

Start with normalization. Most “best practice” relational databases consist of 3NF-5NF tables.

But you can go further - 6NF also exists. An Event is likely either always 5NF or 6NF.

The exact specifications of the design may therefor require 6NF as it’s easier to enforce algorithmically.

This can lead to an explosion of Event models though - but this too is less of a problem if they are also defined algorithmically.

So then - what is the source of truth for the source of truth? Yet Another Relational Model?

LinkML is not quite expressive enough yet to perfectly encapsulate the full breadth of adaptability required.

TypeDB offers a better hint as to what is required…

6.5. Infrastructure Deployment Patterns

DIZZY defers infrastructure decisions by design — but at some point they must be made. A Procedure must run somewhere. A Model must be backed by something. A Command must travel from its emitter to its handler by some mechanism.

The open question is not *whether* to deploy but *how many deployment units to create*. DIZZY’s logical boundaries do not prescribe physical boundaries. A system with eight Command types and twelve Event types could be deployed as a single process, as one service per component, or as any topology in between. All of these are valid. The Feature File makes that topology decision explicit and auditable — not implicit.

6.6. Compaction

DIZZY’s philosophy encourages fine-grained decomposition — one Procedure per Command type, one Projection per event-to-model mapping. Taken to its logical conclusion, a non-trivial system generates dozens of small, independently deployable components. This is the right *logical* structure. It is not always the right *operational* structure.

Compaction is the practice of stitching multiple DIZZY components into a single deployable artifact without violating their logical separation. The domain contracts remain intact. The component boundaries remain auditable. What changes is only the deployment boundary — multiple components share a process, a queue, or a runtime context.

This is not a regression to the monolith problem DIZZY was designed to escape. The monolith problem is that logical boundaries collapse: Procedures call each other directly, Models accumulate mixed concerns, and the component map becomes fiction. Compaction preserves the logical boundaries while making a pragmatic choice about where to draw the operational ones.

7. Conclusion

For the formal specification of DIZZY components, schemas, and code generation pipeline, see the companion *DIZZY Specification*.

8. Appendix

8.1. Common Patterns

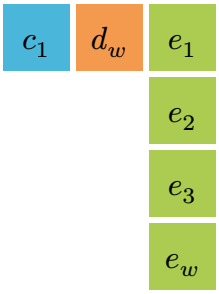
8.1.1. Durable Execution

A procedure can *preform work* if and only if the work has never been started, or in the cases where a previous fault or outage interrupted the runtime.

An explicit failure-as-ended state helps to disambiguate between runtime failures of the procedure and external failures like network interruption, power loss, etc.

For some procedure d_w a durable version d_d is a procedure that wraps d_w such that

1. **Durable Execution Procedure** queries for **Activity Started**
    1. If no **Activity Started**, then emit **Activity Started**
    2. If any **Activity Started**, then query for **Activity Ended** on ID
        1. If no **Activity Ended** - do d_w.
        2. If any **Activity Ended** - return



Symbol	Component	Role
c_1	Command	Start Activity with ID
d_d	Procedure	Durable Execution Procedure
e_1	Event	Activity Started
e_2	Event	Activity Ended (Succeeded)
e_3	Event	Activity Ended (Failed)
e_w	Event	Existing d_w events

8.1.2. Provenance

- Activities
- Entities
- Agents (Human and LLM)

9. Glossary

DIZZY Components

Command. An expression of intent — what a user or policy wants to happen. Commands are ephemeral; they may be rejected, retried, or discarded without consequence. 2, 7, 8, 9, 12, 13, 15, 16, 17, 18, 19

Event. An immutable record of a fact. Events are the single source of truth in a DIZZY system; once written, they cannot be altered or deleted. 2, 7, 8, 9, 10, 12, 13, 14, 15, 16, 17, 18, 19

Model. A queryable view of state, derived from Events via Projections. Models are ephemeral and rebuildable on demand — a cache, not the source of truth. 2, 7, 8, 10, 13, 14, 15, 17, 18, 19, 20

Policy. Reacts to an Event and may emit Commands. Policies encode business rules as reactions to facts, decoupled from the Procedures that produced those facts. 2, 7, 9, 10, 12, 13, 14, 15

Procedure. Handles a Command and may emit Events. Procedures are the site of critical work — where intent meets consequence. 2, 7, 8, 9, 12, 13, 14, 15, 19, 20

Projection. Listens for specific Events and updates one or more Models. Translates the language of facts into queryable state. 2, 10, 12, 13, 14, 15, 17, 19

Querier. Executes a Query against a Model using program code. Keeps database driver logic separate from data definitions and domain logic. 2, 11, 17, 23

Query. A typed data definition for a call and its response — the interface between a Querier and a Model. 2, 10, 11, 12, 13, 14, 15, 16, 17, 18

Architecture

CQRS. Command Query Responsibility Separation. 7

Event Driven. 7

Event Sourcing. 7

Feature File. A structured declaration of every component the system requires: which Commands exist, which Procedures handle them, which Events are produced, which Policies react, and which Projections and Queriers serve the system. The output of a workshopping session and the input to the generation pipeline. 15, 16, 17, 19

Interstitial Infrastructure. The wiring between DIZZY components — message queues, database connections, deployment topology. Kept deliberately separate from domain logic so any infrastructure choice can change without touching business rules. 1, 2, 11, 16

Bibliography

BRANDOLINI, Alberto, 2021. *Introducing EventStorming*. Online. Leanpub. Available from: https://leanpub.com/introducing_eventstorming

BROOKS, Frederick P., 1986. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*. 1986. Vol. 19, no. 4, p. 10–19.

Reprinted in \textit{The Mythical Man-Month: Essays on Software Engineering}, Anniversary Edition, Addison-Wesley, 1995

CONWAY, Melvin E., 1968. How Do Committees Invent?. *Datamation*. Online. April 1968. Available from: http://www.melconway.com/Home/Committees_Paper.html

DIJKSTRA, Edsger W., 1972. *The Humble Programmer*. 1972.

ACM Turing Lecture

DIJKSTRA, Edsger W., 2000. The Threats to Computing Science. . Online. 2000. Available from: https://www.cs.utexas.edu/~EWD/transcriptions/EWD13xx/EWD1305.html

EVANS, Eric, 2003. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley.

FEATHERS, Michael, 2004. *Working Effectively with Legacy Code*. Prentice Hall.

FOWLER, Martin, 2005. Event Sourcing. . Online. 2005. Available from: https://martinfowler.com/eaaDev/EventSourcing.html

FOWLER, Martin, 2011. CQRS. . Online. 2011. Available from: https://martinfowler.com/bliki/CQRS.html

KLEPPMANN, Martin, 2017. *Designing Data-Intensive Applications*. O'Reilly.

KORZYBSKI, Alfred, 1933. *Science and Sanity: An Introduction to Non-Aristotelian Systems and General Semantics*. Online. 5. Institute of General Semantics. ISBN 0-937298-01-8. Available from: https://ilam3d.wordpress.com/wp-content/uploads/2010/12/alfred-korzybksi-science-and-sanity.pdf

LEROY, Jonny and SIMONS, Matt, 2010. Dealing with Creaky Legacy Platforms. *Cutter IT Journal*. Online. December 2010. Available from: https://jonnyleroy.com/2011/02/03/dealing-with-creaky-legacy-platforms/

Republished online February 3, 2011

PERCIVAL, Harry and GREGORY, Bob, 2020. *Architecture Patterns with Python*. Online. O'Reilly. Available from: https://www.cosmicpython.com/

SKELTON, Matthew and PAIS, Manuel, 2019. *Team Topologies*. IT Revolution Press.

STANDISH GROUP, 2015. *CHAOS Report 2015*. Online. technical report. Available from: https://www.standishgroup.com/sample_research_files/CHAOSReport2015-Final.pdf