ChatGPT

# TypeScript Code Review for the dacument Repository

## Overview of the Codebase

The **dacument** repository implements a schema-driven CRDT document with role-based ACLs and signed operations. It is written in TypeScript with a modular structure: core CRDT structures are implemented in separate classes (e.g., `CRArray`, `CRMap`, `CRRecord`, etc.), and the main `Dacument` class orchestrates these with networking/ACL logic. Overall, the code demonstrates a solid grasp of TypeScript's type system (using strict mode) and employs advanced patterns like mapped types and `Proxy` for dynamic behavior. However, there are several areas where type safety could be tightened and code clarity improved. Below, we identify key issues and suggest improvements in terms of type usage, misuse of TS features, code structure, and best practices.

## Type Safety: Broad Types and `any` Usage

**Dynamic Proxies and `any`:** One notable pattern is the use of JavaScript Proxies to provide convenient dynamic access to CRDT collections (e.g., treating a CRDT record as a normal object with properties). In these cases, the code often resorts to `any` to satisfy the TypeScript compiler. For example, the `CRRecord` constructor returns a `Proxy` of `this` cast to `any` [1]. This means the `CRRecord` instance is essentially untyped (as `any`) at runtime for proxying purposes. Similarly, the `Dacument` class returns a Proxy of itself cast to `this` (which effectively masks dynamic field accesses behind the static `DocFieldAccess` type) [2]. While this approach works, it bypasses compile-time type checking. In `CRRecord/class.ts`, for instance, writing to arbitrary properties is done via `(crdt as any)[prop] = value` [3] during initialization of record fields. These uses of `any` sacrifice type safety for flexibility – the compiler can no longer catch type errors on those operations.

**Broad Union Types for CRDT Fields:** The internal representation of document fields uses very broad types. The `FieldState.crdt` is a union of all CRDT classes with their generic parameters as `any` [4]. This design (using `CRArray<any> | CRMap<any, any> | CRSet<any> | ...`) means that once a field's CRDT is stored, the precise value types are erased to `any`. For example, even if a schema defines a field as a set of strings, the `FieldState.crdt` will be `CRSet<any>`. The code compensates by performing runtime type checks (e.g., `isValueOfType`) before writes and by mapping to high-level *view* types for public access. However, this broad typing could allow misuses internally; e.g., calling a method with a wrong value type would not be caught by the compiler on the `any` type. Ideally, the type of each field's CRDT would be parameterized by the schema, so that (for instance) a `CRSet<any>` could be a `CRSet<string>` for a string set field at compile time. As an improvement, **consider using generics in `FieldState` to carry the expected value type**, or storing more specific types in separate fields. This would allow removing many casts when using these CRDT instances.

**Record Fields and Index Signatures:** The **Record** type fields (OR-Records) are handled with maximum flexibility but minimal static typing. In the public API, a record field is typed as `Record<string, T>` (via `RecordView<T>` in `DocFieldAccess`) [5], meaning any property name yields a value of type

`T`. Internally, however, the `CRRecord` class does not declare an index signature; instead it uses a Proxy to trap property access. Because no index signature exists on `CRRecord`, the code must use casts to `any` for property reads/writes (as noted above) and the compiler cannot enforce, for example, that only values of the correct type are assigned. One way to improve safety would be to **give `CRRecord` an index signature** (e.g. `[prop: string]: V | undefined`), or at least to define an interface for its dynamic behavior. With an index signature, TypeScript would allow `crdt[prop] = value` where `value: V` without an explicit cast. There is a potential downside – existing fixed properties of `CRRecord` (like its methods and internal fields) would need to be compatible with this signature. In this case, since `V` defaults to `unknown`, those known properties would still type-check (any specific type is assignable to `unknown`). Defining an index signature could eliminate `(crdt as any)` in many places and let the compiler check that assigned values match the field's declared type.

**Use of `unknown` vs `any`:** The code appropriately uses `unknown` in several places to represent arbitrary JSON-like values (for example, `JsTypeMap` maps the `"any"` JSON type to TypeScript `unknown` [6]). This is good practice, as `unknown` forces explicit type checking or casting before use. However, there are instances where `unknown` values are immediately cast to `any`, which negates the safety. For example, map and set schema definitions accept a user-provided `key` function of type `(value: JsTypeValue<T>) => string`, but in the schema they are stored as `(value: unknown) => string`. When initializing CRDTs, the code casts these back to the specific type, e.g., `crdt.key as (value: unknown) => string` [7]. This is essentially an unchecked cast – if the provided function's signature doesn't actually match the expected type, it could cause runtime errors. To improve this, consider **carrying generic parameters for key functions in schema types**. For instance, `MapSchema` might be generic in the key type `K` in addition to the value type `T`, so that the key function can be typed as `(value: K) => string` throughout. This would propagate to `CRMap<K,V>` creation without the need for `as` casting. Overall, try to keep values as `unknown` until they are *safely* narrowed, rather than casting to `any`.

**Event Listener Map Casting:** The design for event listeners in `Dacument` uses a single `Map` to store listeners for various event types. The map value type is a set of function accepting **any** Dacument event (`DacumentEventMap[keyof DacumentEventMap]`) [8]. Consequently, adding or removing a listener requires casting the specific event handler to that broad type [9]. This is a minor type-safety compromise – it relies on the programmer to ensure the cast is correct. A more type-safe approach would be to store the listeners in a structure keyed by event type with appropriately typed callback sets. For example, the map's value could be a union type or a discriminated type that retains the event mapping. Another approach is to define an overload or separate method for each event type, though that can be verbose. At minimum, **document the cast clearly** or use a comment to indicate why it's safe (because the map segregates by key). This will aid maintainers in understanding that the cast is intentional. In general, the code does a reasonable job keeping the cast local (only in the add/remove methods), but the pattern could be refactored to avoid `any`-casting altogether.

## Misuse or Risky Use of TypeScript Features

**Casting Away Type Safety:** As discussed, the heavy reliance on `as any` in proxy handlers and certain initializations is a red flag from a TypeScript perspective. While dynamic proxies are inherently hard to type, it's worth examining if the same functionality could be achieved with less casting. For instance, the `CRArray` class uses a Proxy but casts it to `this` (of type `CRArray<T>`) on return [10]. This means that at compile time, a `CRArray<T>` instance doesn't actually reflect that you can do array-indexed access (`arr[0]`) – TS will not allow `CRArray` to be indexed by number because no index signature is declared. So ironically, the Proxy provides runtime support for `arr[0]`, but the compiler still disallows it. If the intention was to let users treat `CRArray` like a normal array, consider **implementing**

`ArrayLike<T>` **or providing an index signature** for numeric indices. For example, `interface CRArray<T> extends ArrayLike<T> { … }` could let TS know that `0 .. length-1` indices are valid keys. As it stands, the cast `as this` simply suppresses any compile-time complaint about the Proxy return, without actually extending type information for numeric indexing. It might be acceptable (users can use `at()` and iteration as provided), but it's something to be aware of – the cast is hiding a mismatch between runtime behavior and static type.

**Returning** `unknown` **from Methods:** In the *view* types (like `TextView`, `ArrayView`, etc.), some mutation methods are typed to return `unknown` or a very generic type. For example, `TextView.insertAt` returns `unknown` [11], and `SetView.add` returns `unknown` [12]. This seems to be a workaround for the fact that the underlying CRDT methods return their own instances (`this`), which in the context of the high-level view is not meaningful (e.g., `CRSet.add` returns the CRSet itself, but you don't want to return the internal CRSet to the caller). By marking it as `unknown`, the API discourages the user from chaining or relying on the return. While this is functionally correct, it's a bit of a *code smell*: returning `void` (or a more specific type) might communicate intent better than `unknown`. For instance, one could define `SetView.add` to return `SetView<T>` or simply `void`. Returning `unknown` essentially forces the user to ignore the return or do their own cast, which isn't ideal. **Recommendation:** Return `void` for methods where the return value isn't intended to be used. This makes the API clearer – e.g., `doc.tags.add(x)` would be a `void` operation to add an element, instead of an `unknown` that cannot be used. (If chaining is desired, an alternative is to return the view itself, enabling fluent style. But in this design, views are not the same object as the CRDT, so chaining was intentionally disabled.)

**Inconsistent Return in** `commitRecordMutation`**:** A subtle issue related to the above is that the internal helper `commitRecordMutation` (which wraps record-field mutations in the signing logic) does not return a value in one branch [13], whereas the analogous helpers for array, set, map do return the result of the operation [14] [15]. In `commitRecordMutation`, if no patches are generated (meaning no change occurred), it `return`s early without a value [16]. This causes the function's return type to include `undefined`. In contrast, `commitArrayMutation` and others always return `result` (or propagate it), so they preserve the return type of the underlying CRDT operation (e.g., returning a popped element, or a boolean from delete). The inconsistency could be accidental. For instance, if you call `delete` on a record field via the view (using the `delete` operator on a property), a `false` result (meaning the property wasn't present) might be lost and come out as `undefined`. From a type perspective, this indicates either a missing `return result` or an intentional choice to ignore the result for records. It's worth reviewing this for correctness. If it's a bug, adding `return result;` in that branch would align it with the others. If it's intentional (perhaps record operations are always supposed to succeed in producing a patch when they actually change something), ensure the signature reflects that (maybe make it return `void` if the result is truly never used). In any case, such inconsistencies can confuse maintainers and are caught by TypeScript only if you have `noImplicitReturns` on (which you might consider enabling for stricter checking).

**Using** `private` **vs.** `#` **for truly internal methods:** The code uses `private` class members, which is fine for compile-time. One thing to note, though, is that because `CRRecord.set` is a private method, it can still be accessed via Proxy since at runtime it exists on the object. Indeed, the `createRecordView` Proxy simply forwards any unknown property get to the underlying `CRRecord` [17]. This means a user could accidentally access internal methods. For example, `doc.myRecordField.set` will return the `CRRecord.set` function (since `"set"` is not a defined property on the view's empty target, the get trap falls through to `readCrdt()`). If called, it would mutate the CRDT directly without proper signing – definitely not intended. Because the schema disallows fields named `"set"` at the Dacument level (by reserving property names found on the

Dacument instance/prototype) [18], one normally wouldn't have a record field named `set`. But this collision happens with the CRRecord's own method. To mitigate this, consider a couple of strategies: (1) Use ES2022 `#private` fields/methods for truly internal functions like `CRRecord.set` so they aren't accessible even via Proxy (this requires down-compiling if targeting older runtimes, but Node 18+ should support them); or (2) adjust the Proxy's get trap to explicitly guard against exposing certain internal names. The second approach is complex – essentially you'd filter out method names like `"set"`, `"delete"`, etc. – so using `#` private might be cleaner if you want runtime privacy. This is a design decision: it might not be critical if you assume users won't go looking for non-documented properties, but it's a loophole in the encapsulation.

## Code Clarity and Maintainability

**Repository Structure:** The separation of CRDT implementations into their own modules (with tests and README notes for each) is a positive for modularity. Each CRDT class (`CRArray`, `CRSet`, `CRMap`, `CRRecord`, `CRText`, `CRRegister`) handles its own logic for merging operations and maintaining state. The `Dacument` class then composes these. This structure is logical and makes the codebase easier to navigate. One structural concern, however, is the size of `src/Dacument/class.ts` – at over 3000 lines, it is quite large [19] [20]. Large files can hinder maintainability, especially when they mix concerns. In `Dacument/class.ts`, we see a mix of responsibilities: schema initialization, ACL enforcement, networking (merging ops, emitting events), and even some cryptography (token signing/verification via imported helpers). While splitting it up might not be trivial (because many functions share private state), consider whether some parts can be refactored into separate modules or classes. For instance, the ACL logic could potentially live in its own class (the code already uses `AclLog` for some history tracking). The event emitter functionality might be abstracted or at least visually separated. Even without physically splitting the file, **adding more internal documentation and sectioning** would help – e.g., use comments or JSDoc to delineate "// Field view creation functions", "// Operation commit helpers", "// Merge/apply logic", etc. This makes it easier for a new contributor to find relevant sections in the monolithic class.

**Comments and Documentation:** The code does include some comments (especially in the README and CRDT README files explaining the high-level behavior). However, the in-code documentation could be expanded. Critical functions like `createFieldView`, `shadowFor`, and `applyRemotePayload` have complex logic with many cases. Adding JSDoc comments on these functions, describing their purpose and preconditions, would greatly aid understanding. For example, documenting that `shadowFor` creates a transient CRDT copy for mutation could help readers grasp why it's needed. TypeScript interfaces/types could also benefit from comments – e.g., clarifying what `ResetStateInfo` or `AclAssignment` represent. Given that this library deals with security (signatures, roles), clarity in code is as important as type safety. Consider adding explanatory comments for non-obvious type manipulations or casts (especially where `as any` is used out of necessity). A one-line comment like "// Cast to any needed for Proxy, no way to type dynamic props here" can go a long way in code reviews.

**Consistency and Naming:** Overall, naming in the code is clear and consistent (e.g., `applyNodePayload`, `deleteNodeStampsByField` – while lengthy – precisely describe their role). One area to double-check is consistency of similar operations across CRDTs. For instance, `CRArray` implements many array-like methods (`push`, `pop`, `map`, etc.) and the corresponding view methods call these. Ensure that any method provided in the view exists on the CRDT and vice versa. We saw that `CRArray` does implement `.map`, `.filter`, etc., using an `alive()` helper to work on a materialized array [21] [22] – that's good. The `ArrayView.slice` simply calls the CRDT's `slice` (which returns a new array of values) [23]. The only minor clarity issue is the use of `(thisArg as never)` in those implementations [24], presumably to satisfy type requirements for

`Array.prototype.map` and friends. Casting `thisArg` to `never` is an unusual pattern – likely done to hush compiler complaints about `unknown` (since `Array.map` expects the second argument to be of type `any` or a specific context). A clearer approach could be using `undefined` or `void 0` if no `thisArg` is provided, or simply casting to `any`. Casting to `never` works because `never` is assignable to everything (being the bottom type), but it's semantically confusing. It doesn't affect runtime, but from a maintainability perspective, a brief comment or using a more common cast (`as any`) would be preferable here.

**Testing and Type Expectations:** The test files are written in plain JavaScript (`.test.js`), which means they don't directly validate TypeScript typings. Be cautious that some type-related issues might not surface in runtime tests. For example, a test might do `doc.someRecord.nonexistentProp` and expect `undefined`, which will work, but TypeScript (with the broad `Record<string,T>` type) wouldn't flag it either. If possible, writing a few tests in TypeScript or using `tsd` (TypeScript Definition Testing) could help ensure the public typings behave as intended (e.g., you could test that `doc.title = 123` fails to compile if `title` is a string register). This is more of a best-practice suggestion: since the library is distributed with type definitions (`"types": "dist/index.d.ts"` in package.json), ensuring those types are strict and correct is important. Some of the suggestions above (like narrowing `any` usages or adding index signatures) would make the public API safer in the type definitions consumers get.

## Compiler Options and Best Practices

The `tsconfig.json` shows that **strict mode is enabled**, which is excellent (it enforces `noImplicitAny`, `strictNullChecks`, etc.) [25]. A few additional compiler flags could be considered for even stricter checking:

- `noUnusedLocals` **and** `noUnusedParameters`**:** Enabling these can catch leftover variables or parameters that are never used, which helps keep the code clean. This doesn't appear to be a big issue in the code (we didn't spot obvious unused variables), but it's helpful for future maintenance.

- `noImplicitReturns`**:** This would have flagged the `commitRecordMutation` issue where not all code paths return a value. It's useful to ensure functions either always return or never. If this flag was off (likely it was, since strict mode doesn't include it by default), consider enabling it to catch similar cases.

- `exactOptionalPropertyTypes`**:** This flag treats optional properties (`foo?: string`) more strictly with respect to `undefined`. In this codebase, it might highlight if an optional property is being set to `undefined` explicitly or if there's an assumption that optional means "maybe undefined". For example, `field.view?: unknown` is always either a value or `undefined`; using this flag could prevent incorrect assignments. It's an edge case, but worth mentioning if aiming for maximum strictness.

- `skipLibCheck`**:** Currently set to true [25], which is understandable to avoid noise from dependencies. Be aware that if any dependency (like `bytecodec` or `zeyra`) has incorrect type definitions, `skipLibCheck` will mask those errors. In a security-sensitive library, you might actually want to type-check those dependencies or at least stay vigilant. Not a required change, but something to weigh (most projects leave it true for faster builds).

- **Target and Module:** Using ES2022 and NodeNext module resolution is up-to-date, which is good for modern syntax. Just ensure that when publishing, the output is compatible with the consumers' environment (the config outputs ESM to `dist/`). It's mentioned that Node >= 18 is required, so this is consistent.

- **Declaration emit:** The config generates declaration files (`"declaration": true`), which is good for consumers. After making any type changes (like those suggested to reduce `any`), double-check the `.d.ts` output to ensure the public API remains user-friendly.

## Summary of Key Issues and Suggestions (Actionable)

To wrap up, here are the high-impact issues identified along with **actionable suggestions** for each:

- **Excessive use of `any` in Proxies:** Places like `CRRecord` and `Dacument` use `as any` to enable dynamic property access [1] [3]. *Suggestion:* Introduce stricter types where possible (e.g., index signatures or generic parameters for `CRRecord`) to eliminate these casts, or encapsulate them in one place with clear comments if unavoidable.

- **Broadly typed field state:** Internally, all CRDT fields are stored as `any` (e.g., `CRMap<any, any>` in `FieldState` [4]). *Suggestion:* Refactor `FieldState` to carry type parameters for the field's value (perhaps keyed by field name via a mapped type). This is an involved change, but it would allow the compiler to catch type mismatches in field operations and reduce runtime type checking overhead.

- **Type assertions for schema key functions:** The code uses `options.key as (value: unknown) => string` in multiple schema factory functions [7]. *Suggestion:* Change the schema types to preserve the generic type of the key function. For example, if a set holds numbers, the key function's parameter should be `number`, not `unknown`. Propagate that type to `CRSet`/`CRMap` construction to avoid casting. This will make the API safer for users providing custom key functions (they'll get compile-time validation of the function signature).

- **Event listener typing and casting:** Currently all event listeners are managed in one map requiring casts when adding/removing [9]. *Suggestion:* Use a more type-safe approach for event listeners. One option is a separate map or even separate methods per event type (e.g., `onChange(cb: (ChangeEvent) => void)`). If sticking with the current design, add comments to explain the cast is safe, or wrap the casting logic in a utility function to abstract it away. This reduces the chance of misuse.

- **Return types of mutation methods (views vs CRDTs):** Some view methods return `unknown` to mask internal returns [11]. *Suggestion:* Change these to `void` (when no meaningful result for the user) or to a defined type. For example, `doc.tags.add(x)` could return `void` instead of `unknown` (since the user doesn't need the internal CRSet). This makes the API cleaner and avoids confusion.

- **Large `Dacument` class size:** The core class is very long, mixing multiple concerns. *Suggestion:* Refactor logically: e.g., move cryptographic helpers (`signToken`, etc.) to a separate module (they already exist in `crypto.js`), possibly move event emission or ACL logic to helper classes, or at least break up the file with regions and comments. Smaller, focused functions with descriptive names will also help (the code already does this to an extent, but could be improved

by extracting repeated patterns like the nearly identical `commitXMutation` functions – maybe a generic helper factory function could reduce duplication).

- **Compiler rigor:** To catch subtle issues like inconsistent returns or unhandled cases, consider enabling additional strict flags (`noImplicitReturns`, `noUnusedParameters`, etc.). *Suggestion:* Turn on `noImplicitReturns` to catch functions that don't return in all code paths (e.g., it would flag the `commitRecordMutation` early return without value). Also, periodically compile with `skipLibCheck:false` in CI (or locally) to ensure dependent types are sound, since security depends on upstream types too.

Implementing these suggestions will improve the type safety and maintainability of **dacument**. In particular, reducing reliance on `any`/casting will catch more errors at compile time and make the intent clearer to contributors. Despite the few issues noted, the codebase adheres to modern TypeScript best practices in many areas (use of `unknown`, mapped types for schema->value mapping, and strict typing of the public API). With the above refinements, **dacument** can achieve an even higher level of type safety and clarity, benefiting both its developers and users. Good luck with the improvements!

---

1 class.ts
https://github.com/jortsupetterson/dacument/blob/70ae75e159dc9fe631e4408ad081aff8ae591acd/src/CRRecord/class.ts

2 3 4 7 8 9 13 14 15 16 17 18 19 20 class.ts
https://github.com/jortsupetterson/dacument/blob/70ae75e159dc9fe631e4408ad081aff8ae591acd/src/Dacument/class.ts

5 6 11 12 23 types.ts
https://github.com/jortsupetterson/dacument/blob/70ae75e159dc9fe631e4408ad081aff8ae591acd/src/Dacument/types.ts

10 21 22 24 class.ts
https://github.com/jortsupetterson/dacument/blob/70ae75e159dc9fe631e4408ad081aff8ae591acd/src/CRArray/class.ts

25 tsconfig.json
https://github.com/jortsupetterson/dacument/blob/70ae75e159dc9fe631e4408ad081aff8ae591acd/tsconfig.json