



WELCOME

A Leak in the Shell

How Refactoring Autocomplete Broke Us
and How We Fixed It



Anna Henningsen
Staff Engineer @ MongoDB



Who am I?

- Anna Henningsen
- she/her
- Staff Developer @ MongoDB
- Former full-time Node.js core contributor

`@addaleax.bsky.social`

`https://addaleax.net/jsnation2026.pdf`



The Setting





Our CLI

```
$ mongosh
```

```
Connecting to:      mongodb://127.0.0.1:27017/
```

```
[...]
```

```
> db.test.findOne({ orderId: 42 })
```

```
{  
  _id: ObjectId('6a26cc4068e98f690fbb3535'),  
  orderId: 42,  
  customerId: 123  
}
```

```
> db.test.findOne({ orderNumber: 42 }).custo<TAB>
```

WE'VE HAD FIRST AUTOCOMPLETE, YES



BUT WHAT ABOUT SECOND AUTOCOMPLETE?



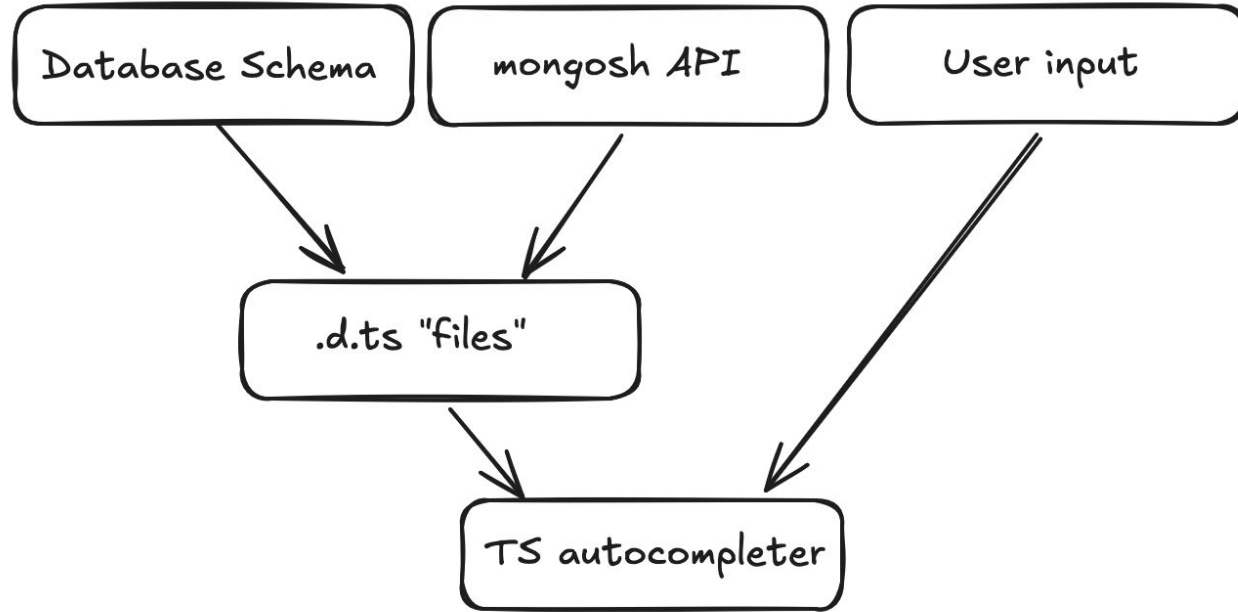


Let's come up with a plan

```
function unpause() {  
  if (!paused) return;  
  paused = false;  
  let entry;  
  const tmpCompletionEnabled = self.isCompletionEnabled;  
  while ((entry = Array.prototype.shift.call(self.isCompletionEnabled  
    const { 0: type, 1: payload, 2: isCompletionEnabled } = entry;  
    switch (type) {  
      case 'key': {  
        const { 0: d, 1: key } = payload;  
        self.isCompletionEnabled = isCompletionEnabled;  
        self._ttyWrite(d, key);  
        break;  
      }  
      case 'close':  
        self.emit('exit');  
        break;  
    }  
    if (paused) {  
      break;  
    }  
  }  
  self.isCompletionEnabled = tmpCompletionEnabled;  
}
```



Let's come up with a plan





Aaand ... it works!



... or does it?





Oops (or maybe Oom...)

```
- completes properties of shell API result types
- completes only collection names that do not include control characters
with an auth-required mongod
autocompletion
  ✓ does not complete query operators (959ms)
  ✓ completes the watch method (533ms)
  ✓ completes the version method (2114ms)
  ✓ does not complete legacy JS get/set definitions (923ms)
  ✓ does not complete the getShardDistribution method (2257ms)
  - includes collection names
  ✓ completes JS value properties properly (incomplete, double tab) (1673ms)

<--- Last few GCs --->
[34843:0x80880c000] 260669 ms: Scavenge (interleaved) 4037.9 (4083.7) -> 4027.4 (4119.0) MB, pooled: 0 MB,
[34843:0x80880c000] 261070 ms: Mark-Compact 4037.9 (4119.0) -> 4027.4 (4119.0) MB, pooled: 0 MB, 3
FATAL ERROR: Ineffective mark-compacts near heap limit Allocation failed - JavaScript heap out of memory
----- Native stack trace -----
1: 0x1046ab714 node::OOMErrorHandler(char const*, v8::OOMDetails const&) [/Users/anna.henningsen/.nvm/versions/node/v24.13.1/bin/node]
2: 0x1048aa7e8 v8::internal::V8::FatalProcessOutOfMemory(v8::internal::Isolate*, char const*, v8::OOMDetails const&)
3: 0x104b0a41c v8::internal::Heap::stackC() [/Users/anna.henningsen/.nvm/versions/node/v24.13.1/bin/node]
4: 0x104b0d7f0 v8::internal::Heap::HasLowYoungGenerationAllocationRate() [/Users/anna.henningsen/.nvm/versions/node/v24.13.1/bin/node]
5: 0x104b1ea40 v8::internal::Heap::CollectGarbage(v8::internal::AllocationSpace, v8::internal::GarbageCollectionOptions, v8::internal::HeapCollectorMode) [/Users/anna.henningsen/.nvm/versions/node/v24.13.1/bin/node]
```



How do you even start debugging this

- Minimal reproducible example ...
 - Each test passed on its own 😊



How do you even start debugging this

- Minimal reproducible example ...
 - Each test passed on its own 😊
 - Reduce heap size (here: `npx mocha -n max-old-space-size=1024`)



How do you even start debugging this

- Minimal reproducible example ...
 - Each test passed on its own 😊
 - Reduce heap size (here: `npx mocha -n max-old-space-size=1024`)
- Need to somehow inspect what's on the heap
 - We're lucky - there's `--heapsnapshot-near-heap-limit=1` (and its sibling `--heap-snapshot-on-oom`)



How do you even start debugging this

The screenshot shows the Chrome DevTools Memory tab with a heap snapshot summary. The table lists various object types and their memory usage. The 'Retained Size' column is highlighted in blue.

| Constructor | Distance | Shallow Size | Retained Size |
|-----------------------------|----------|-----------------|-----------------|
| ▶ system / Context ×39,540 | 3 | 2,416 kB 0 % | 918,311 kB 83 % |
| ▶ Function ×173,837 | 3 | 10,403 kB 1 % | 709,320 kB 64 % |
| ▶ Object ×20,656 | 3 | 1,139 kB 0 % | 699,719 kB 63 % |
| ▶ PrettyREPLServer ×19 | 10 | 0.5 kB 0 % | 639,333 kB 58 % |
| ▶ MongoDBAutocompleter ×15 | 3 | 1.0 kB 0 % | 638,592 kB 58 % |
| ▶ Autocompleter ×15 | 3 | 0.8 kB 0 % | 638,584 kB 58 % |
| ▶ NodeObject ×1,160,299 | 3 | 194,930 kB 18 % | 557,041 kB 50 % |
| ▶ Array ×1,431,185 | 3 | 45,799 kB 4 % | 375,019 kB 34 % |
| ▶ (array) ×1,878,628 | 3 | 176,490 kB 16 % | 199,585 kB 18 % |
| ▶ (string) ×662,647 | 3 | 170,706 kB 15 % | 170,706 kB 15 % |
| ▶ SymbolObject ×503,094 | 26 | 68,421 kB 6 % | 157,427 kB 14 % |
| ▶ NodeObject ×365,447 | 9 | 61,395 kB 6 % | 147,811 kB 13 % |
| ▶ Map ×479,616 | 3 | 15,348 kB 1 % | 131,774 kB 12 % |
| ▶ SourceFileObject ×672 | 11 | 123 kB 0 % | 119,280 kB 11 % |
| ▶ (compiled code) ×461,359 | 3 | 51,988 kB 5 % | 101,573 kB 9 % |
| ▶ IdentifierObject ×763,716 | 3 | 91,646 kB 8 % | 92,320 kB 8 % |
| ▶ SourceFileObject ×1,437 | 3 | 263 kB 0 % | 87,922 kB 8 % |

Retainers

| Object | Distance▲ | Shallow Size | Retained Size |
|--------|-----------|--------------|---------------|
| | | | |



How do you even start debugging this

The screenshot shows the Chrome DevTools Memory tab. The top navigation bar includes Connection, Console, Sources, Network, Memory (selected), Performance, and Application. Below the navigation bar, there are icons for memory management and a filter dropdown set to 'Filter by class' with 'All objects' selected. A 'Restore ignored retainers' link is visible. The main content area is divided into two sections: 'Constructor' and 'Retainers'.

Constructor

| Constructor | Distance | Shallow Size | Retained Size |
|---------------------------------------|----------|--------------|----------------|
| system / Context ×39,540 | 3 | 2,416 kB 0% | 918,311 kB 83% |
| ▶ system / Context @4751375 | 6 | 0.1 kB 0% | 194,459 kB 18% |
| ▶ system / Context @11778199 | 13 | 0.1 kB 0% | 107,154 kB 10% |
| ▶ system / Context @14006475 | 15 | 0.0 kB 0% | 107,153 kB 10% |
| ▶ system / Context @18435027 | 13 | 0.1 kB 0% | 106,683 kB 10% |
| ▶ system / Context @15108207 | 15 | 0.0 kB 0% | 106,682 kB 10% |
| ▼ system / Context @15011771 | 13 | 0.1 kB 0% | 106,531 kB 10% |
| ▶ repl :: PrettyREPLServer @9125721 | 10 | 0.0 kB 0% | 106,566 kB 10% |
| ▶ newMongoshCompleter :: () @15944413 | 14 | 0.1 kB 0% | 106,530 kB 10% |

Retainers

| Object | Distance | Shallow Size | Retained Size |
|--|----------|--------------|----------------|
| ▼ context in () @14019111 | 21 | 0.1 kB 0% | 0.1 kB 0% |
| ▼ exit in {<symbol async-repl:evalFinish>, <symbol...} | 20 | 0.0 kB 0% | 1.8 kB 0% |
| ▼ _events in PrettyREPLServer @9125721 | 19 | 0.0 kB 0% | 106,566 kB 10% |
| ▼ this in system / Context @16549509 | 18 | 0.0 kB 0% | 0.0 kB 0% |
| ▼ context in get() @16549513 | 17 | 0.1 kB 0% | 0.1 kB 0% |
| ▼ get_error in {Decimal128, structured... | 16 | 0.0 kB 0% | 17.5 kB 0% |
| ▼ context in system / Context @180597... | 15 | 0.2 kB 0% | 0.6 kB 0% |
| ▼ previous in system / Context @180... | 14 | 0.0 kB 0% | 0.6 kB 0% |
| ▼ previous in system / Context @1... | 13 | 0.1 kB 0% | 13.7 kB 0% |
| ▼ previous in system / Context @... | 12 | 0.1 kB 0% | 15.8 kB 0% |
| ▼ context in () @15247657 | 11 | 0.1 kB 0% | 2.1 kB 0% |
| ▼ origArraySort in system | 10 | 0.1 kB 0% | 13.7 kB 0% |
| ▼ previous in system / C | 9 | 0.1 kB 0% | 15.0 kB 0% |



How do you even start debugging this

- ▶ system / Context @11778199
 - ▶ system / Context @14006475
 - ▶ system / Context @18435027
 - ▶ system / Context @15108207
 - ▼ system / Context @15011771
 - ▶ repl :: PrettyREPLServer @9125721
 - ▶ newMongoshCompleter :: () @15944413
-

ALWAYS HAS BEEN

**WAIT IT'S ALL
SYSTEM / CONTEXT?**





Let's back up a
bit



How does this retain memory?

```
globalThis.a = {  
  b: {  
    c: {  
      reallyLargeBlob: new  
        Uint8Array(10000000)  
    }  
  }  
};
```



How does this retain memory?

```
globalThis.a = {  
  b: {  
    c: {  
      reallyLargeBlob: new  
        Uint8Array(10000000)  
    }  
  }  
};
```

| | | | | | |
|------------------------------------|----|-----------|------|-----------|------|
| ▶ ArrayBuffer x40 | 3 | 3.9 kB | 0 % | 10,027 kB | 00 % |
| ▼ system / JSArrayBufferData x35 | 6 | 10,023 kB | 65 % | 10,023 kB | 65 % |
| system / JSArrayBufferData @100983 | 7 | 10,000 kB | 65 % | 10,000 kB | 65 % |
| system / JSArrayBufferData @69335 | 12 | 8.2 kB | 0 % | 8.2 kB | 0 % |
| system / JSArrayBufferData @100811 | 7 | 8.2 kB | 0 % | 8.2 kB | 0 % |
| system / JSArrayBufferData @67925 | 8 | 4.1 kB | 0 % | 4.1 kB | 0 % |
| system / JSArrayBufferData @70663 | 8 | 0.3 kB | 0 % | 0.3 kB | 0 % |

| Object | Distance▲ | Shallow Size | Retained Size |
|--|-----------|--------------|----------------|
| ▼ backing_store in ArrayBuffer @100981 | 6 | 0.1 kB | 10,000 kB 65 % |
| ▼ buffer in Uint8Array @100979 □ | 5 | 0.1 kB | 10,000 kB 65 % |
| ▼ reallyLargeBlob in {reallyLargeBlob} @7179 □ | 4 | 0.0 kB | 10,000 kB 65 % |
| ▼ c in {c} @100977 □ | 3 | 0.0 kB | 10,001 kB 65 % |
| ▼ b in {b} @71789 □ | 2 | 0.0 kB | 10,001 kB 65 % |
| ▶ a in global @7065 □ | 1 | 7.8 kB | 15,002 kB 98 % |
| ▶ value in system / PropertyCell @71925 | 3 | 0.0 kB | 0.0 kB 0 % |
| ▶ 24 in (Stack roots) @41 | - | 0.0 kB | 2.8 kB 0 % |



How does this retain memory?

```
globalThis.a = {  
  b: {  
    c: {  
      reallyLargeBlob: new  
        Uint8Array(10000000)  
    }  
  }  
};
```

```
▼ backing_store in ArrayBuffer @100981  
  ▼ buffer in Uint8Array @100979 □  
    ▼ reallyLargeBlob in {reallyLargeBlob} @7179 □  
      ▼ c in {c} @100977 □  
        ▼ b in {b} @71789 □  
          ► a in global @7065 □
```



How does *this* retain memory?

```
const reallyLargeBlob =  
  new Uint8Array(10000000);  
  
setInterval(function a() {  
  console.log('done');  
}, 1000);  
  
setTimeout(function b() {  
  reallyLargeBlob.fill(1);  
}, 1);
```



How does *this* retain memory?

```
const reallyLargeBlob =  
  new Uint8Array(10000000);
```

```
setInterval(function a() {  
  console.log('done');  
}, 1000);
```

```
setTimeout(function b() {  
  reallyLargeBlob.fill(1);  
}, 1);
```

```
▼ backing_store in ArrayBuffer @100961  
  ▼ buffer in Uint8Array @100959  
    ▼ reallyLargeBlob in system / Context @7181  
      ▼ context in a() @100965  
        ▼ _onTimeout in Timeout @100967  
          ▼ _idlePrev in TimersList @66647  
            ▼ [1000] in {} @66507  
              ▼ timerListMap in system / Context
```



JS can be more evil than this

```
let a = 2;  
eval('a+a'); // prints ?
```



JS can be more evil than this

```
let a = 2;  
eval('a+a'); // prints ?  
eval('eval("a+a")'); // prints ?
```



JS can be more evil than this

```
let a = 2;  
eval('a+a'); // prints ?  
eval('eval("a+a")'); // prints ?  
eval('eval')('a+a'); // prints ?
```

The actual bugs





The really unnecessarily dumb

```
1168     /**
1169     * Close all open resources held by this REPL instance.
1170     */
1171     async close(): Promise<void> {
1172         return (this.closingPromise ??= (async () => {
1173             markTime(TimingCategories.REPLInstantiation, 'start closing');
1174 +         await this.mongoshRepl.close();
1175             this.agent?.destroy();
```



The really tricky to detect

```
121 // in which the shell-api package lives and not from the context
122 // the REPL (i.e. `db.test.find().toArray() instanceof Array` is
123 if (!hasAlreadyRunGlobalRuntimeSupportEval) {
-   eval(supportCode);
+   contextlessEval(supportCode);
+   hasAlreadyRunGlobalRuntimeSupportEval = true;
126 }
apps 7 this.markTime?.(
```



The Node.js core bug

```
307     module.exports.repl = this;
-   } else if (!addedNewListener) {
-     // Add this listener only once and use a WeakSet that contains the REPLs
-     // domains. Otherwise we'd have to add a single listener to each REPL
-     // instance and that could trigger the `MaxListenersExceededWarning`.
-     process.prependListener('newListener', (event, listener) => {
-       if (event === 'uncaughtException' &&
-         process.domain &&
-         listener.name !== 'domainUncaughtExceptionClear' &&
-         domainSet.has(process.domain)) {
-         // Throw an error so that the event will not be added and the current
-         // domain takes over. That way the user is notified about the error
-         // and the current code evaluation is stopped, just as any other code
-         // that contains an error.
-         throw new ERR_INVALID_REPL_INPUT(
-           'Listeners for `uncaughtException` cannot be used in the REPL');
-       }
-     });
-     addedNewListener = true;
368 +   } else {
369 +     addProcessNewListener();
370 +     this.once('exit', removeProcessNewListener);
371   }
372
```

... and how do
you test this?



GC in JS is unobservable





GC in JS is ~~un~~observable

- `FinalizationRegistry` - tricky to get right, but for testing that's fine!



GC in JS is ~~un~~observable

- `FinalizationRegistry` - tricky to get right, but for testing that's fine!
- `v8.queryObjects()` and `v8.getHeapSnapshot()` let you actually inspect your application's JS heap in real time!



GC in JS is ~~un~~observable

- `FinalizationRegistry` – tricky to get right, but for testing that's fine!
- `v8.queryObjects()` and `v8.getHeapSnapshot()` let you actually inspect your application's JS heap in real time!
 - Obvious caveat: These are *diagnostic* tools only



So, let's remember...

- Do not over-optimize against these patterns – keep them in *mind* and know when to look for them!



So, let's remember...

- Do not over-optimize against these patterns – keep them in *mind* and know when to look for them!
- Heap snapshots are an incredibly helpful tool



So, let's remember...

- Do not over-optimize against these patterns – keep them in *mind* and know when to look for them!
- Heap snapshots are an incredibly helpful tool
- Be empowered to fix Node.js bugs that you run into!

The AI angle





The AI angle

Why it's retained (the mechanism)

The buffer isn't referenced directly by anything long-lived. It lives as a variable (`reallyLargeBlob`) inside a lexical scope — `system / Context` id 7177. Two closures capture that same scope:

```
closure "b" (id 7175) - _onTimeout of Timeout id 7179
closure "a" (id 81439) - _onTimeout of Timeout id 81441
```

Both are registered as the `_onTimeout` handler of `Timeout` objects sitting in Node's timer structures, which are anchored in **global handles** (GC roots). As long as either timer remains scheduled, its callback is reachable; the callback keeps the shared `Context` alive; and the `Context` keeps the 10 MB blob alive. This is the classic trap: a function keeps a large variable alive **even if that function never uses it**, simply because they share a scope, and the function outlives where you'd expect the data to die.



The AI angle

- ... but *real* heap snapshots are large



The AI angle

- ... but *real* heap snapshots are large
- ... but *real* heap snapshots contain *your* data



The AI angle

- ... but *real* heap snapshots are large
- ... but *real* heap snapshots contain *your* data
- ... but *you* should still understand what your application does!



Thank you for
your time!

