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>
```

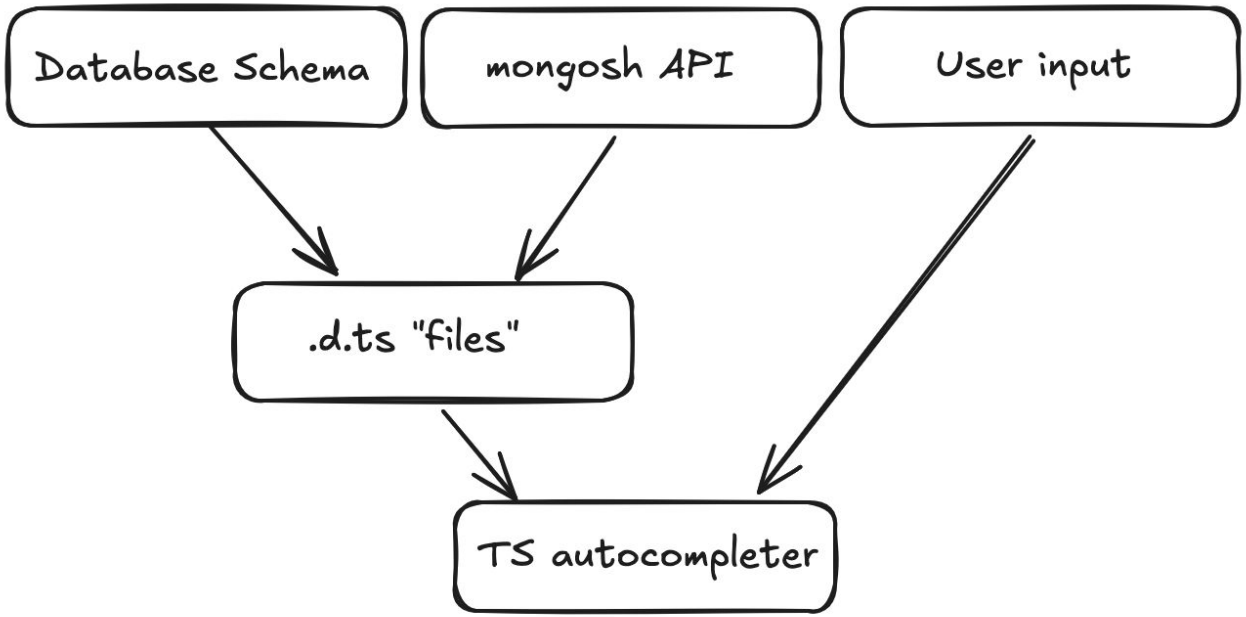

WE'VE HAD FIRST AUTOCOMPLETE, YES
BUT WHAT ABOUT SECOND AUTOCOMPLETE?

Let’s come up with a plan

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

```
isCompletionEnabled
isRecoverableError
isObjectLiteral
isErrorStackTraceLimitWritable
onSigCont
SideEffectFreeRegExpPrototypeSymbolReplace
SideEffectFreeRegExpPrototypeSymbolSplit
sendInspectorCommand
runInThisContext
historyConfig
inspect
listenerCount
```

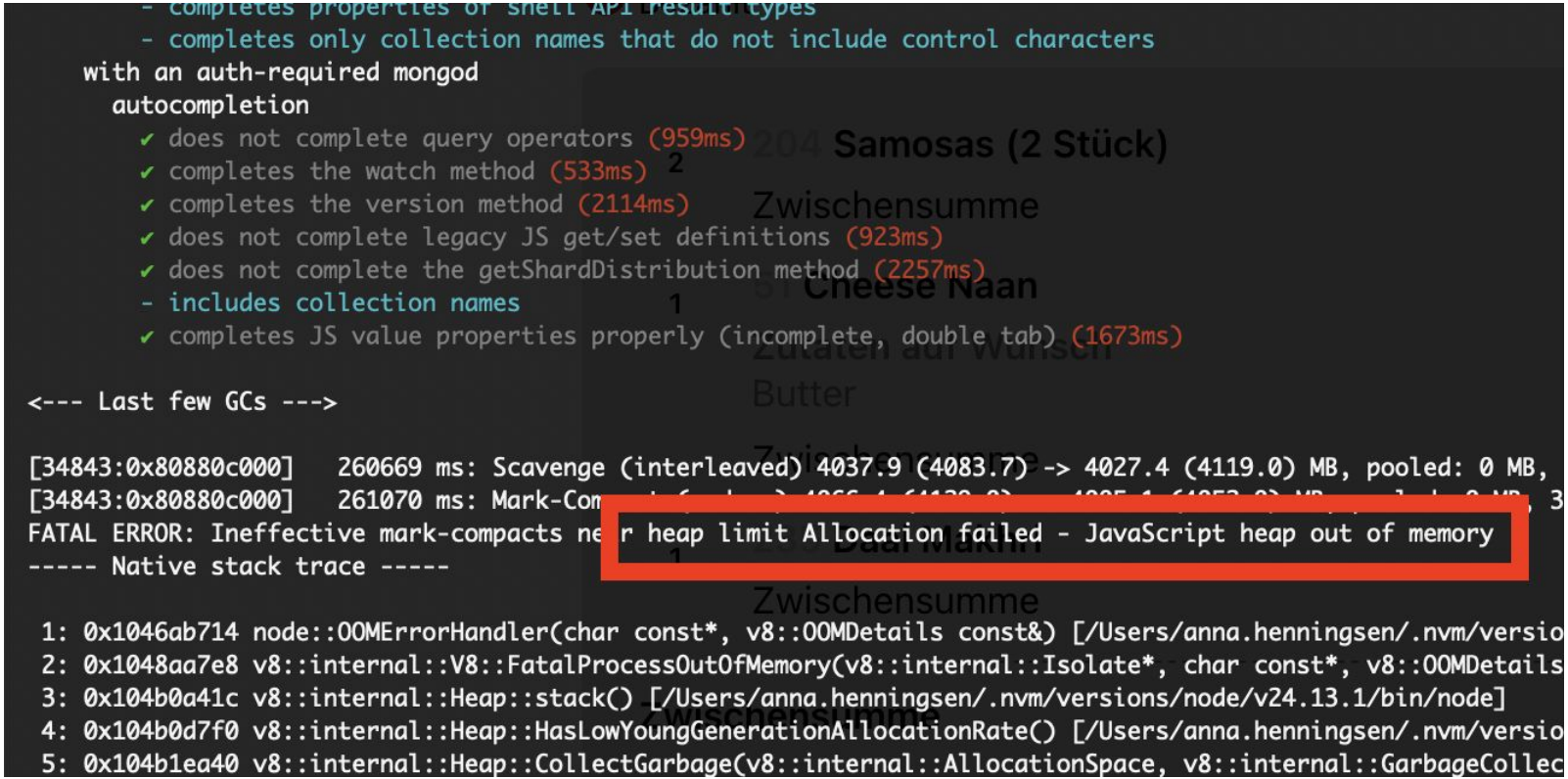
Let’s come up with a plan



Aaand … it works!

... or does it?

Oops (or maybe Oom...)



How do you even start debugging this

- Minimal reproducible example …
  - Each test passed on its own 🥲

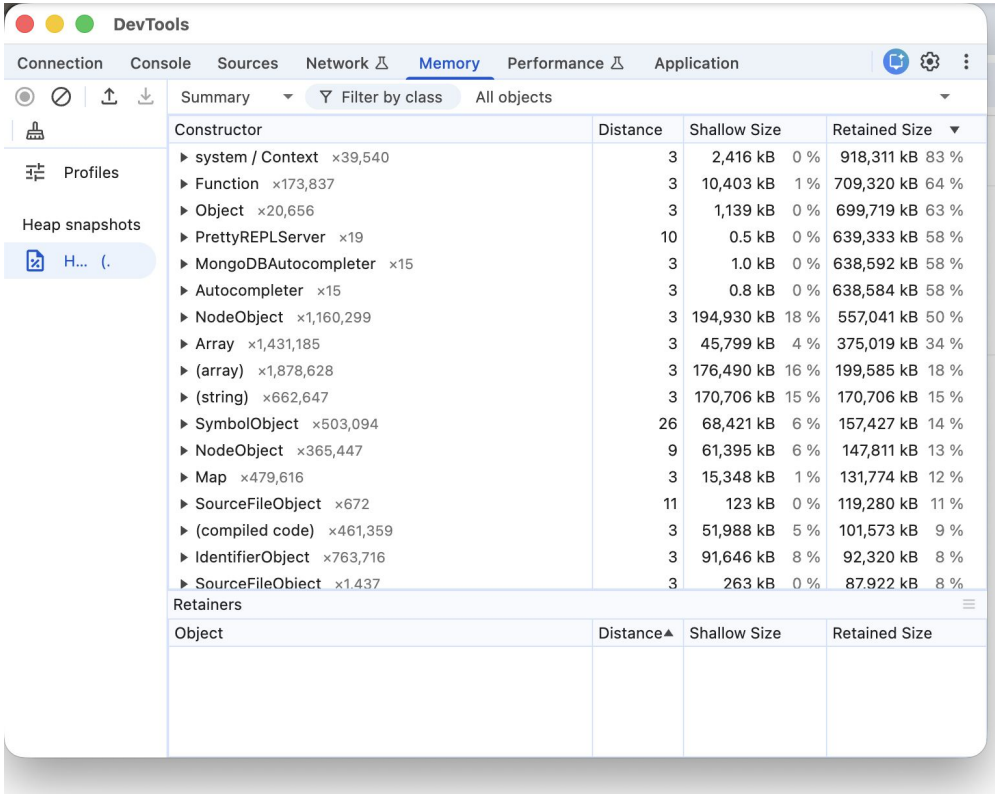
How do you even start debugging this

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

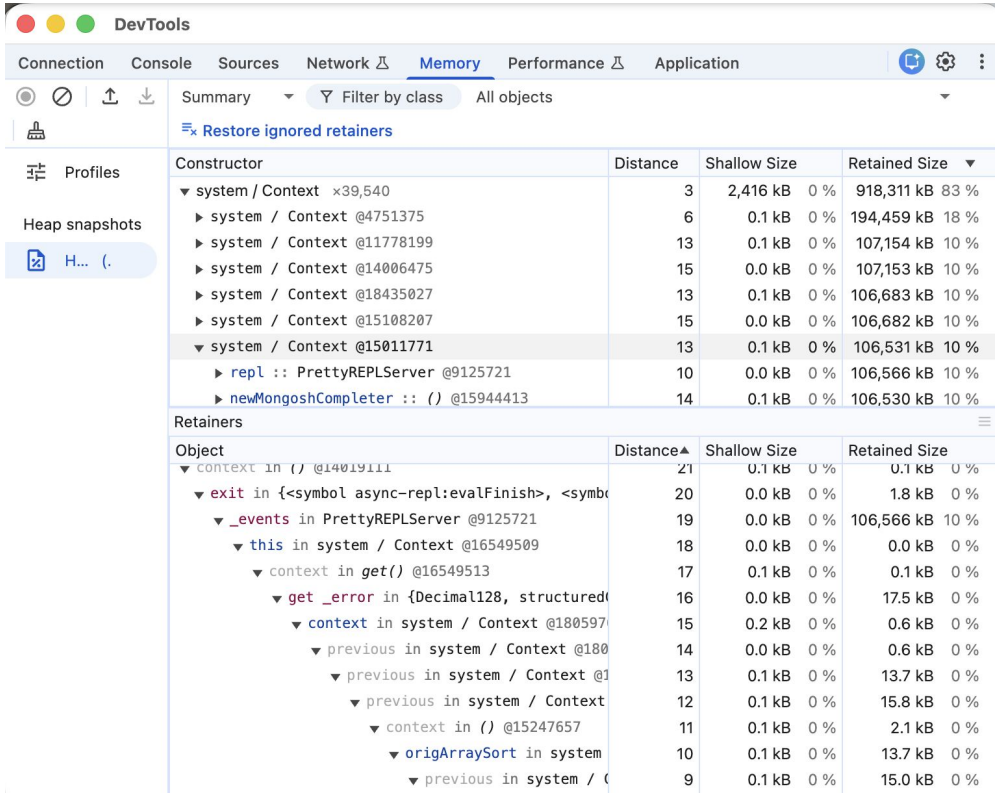
How do you even start debugging this

- 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



How do you even start debugging this



How do you even start debugging this


ALWAYS HAS BEEN
WAIT IT'S ALL
SYSTEM / CONTEXT?

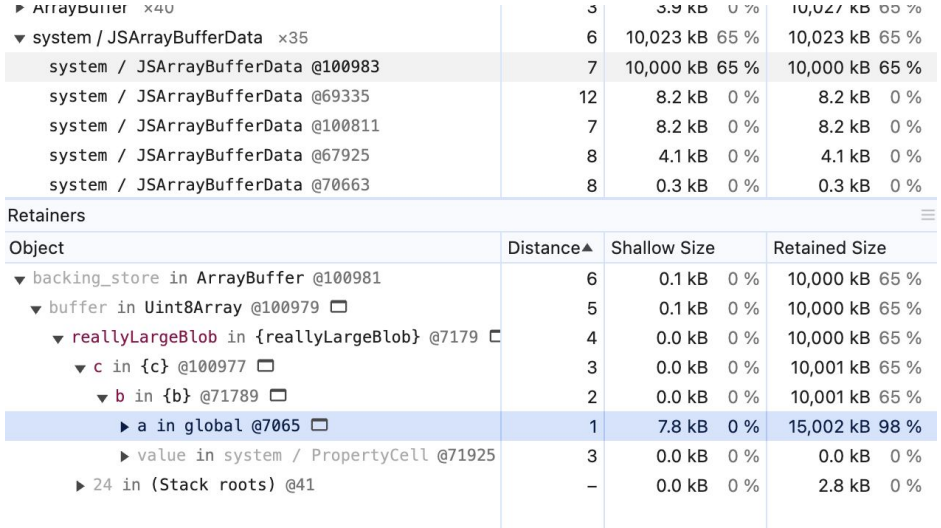
Let’s back up a bit

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)
    }
  }
};
```



How does this retain memory?

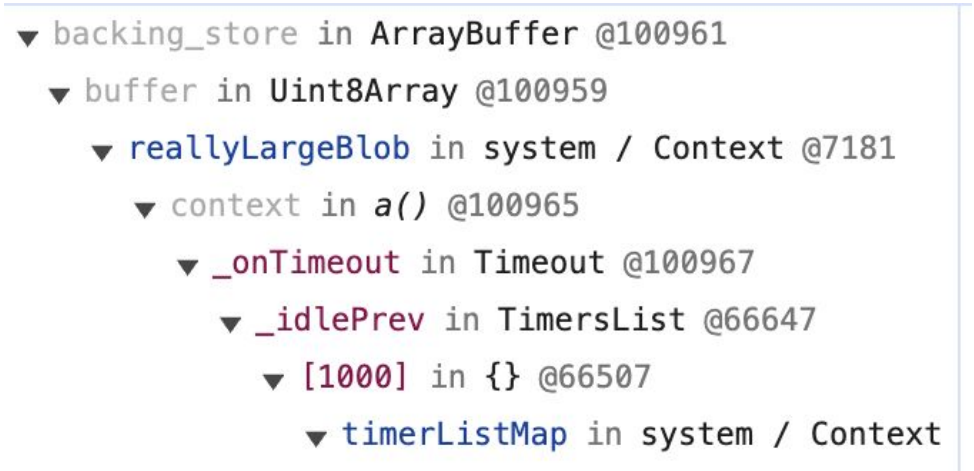
```
globalThis.a = {
  b: {
    c: {
      reallyLargeBlob: new
        Uint8Array(10000000)
    }
  }
};
```

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);
```



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);
124 +           contextlessEval(supportCode);
125 +           hasAlreadyRunGlobalRuntimeSupportEval = true;
126         }
127         this.markTime?.(
```

The Node.js core bug

```
367       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     }
```

… 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!