WEKA

Tuesday, 19st August 2025

D at WEKA, World’s Fastest Data Platform

Eyal Lotem

WEKA

WEKA

Background

- World's fastest, distributed, parallel file system and data platform
- Founded in 2014
- Software-only
- Solves some of the toughest technical challenges
- Exabyte scale, terabytes/sec at sub-millisecond latencies
- The most interesting customers in the world!

WEKA

Intro

■Weka uses D for many domains:

- Networking
- Hardware abstraction
- Clustering
- Filesystem logic
- Software RAID
- RAFT

WEKA

Intro

■Weka uses D for many domains:

- Networking
- Hardware abstraction
- Clustering
- Filesystem logic
- Software RAID
- RAFT

- And much much more!

D @ Weka

RPC

- Virtually all network traffic is remote procedure calls (RPCs)
- Efficiency is critical
- Binary protocol
- Based on homegrown performance-optimized network stack

D @ Weka

RPC

- Declare via a regular D interface
- Efficient RPC client & server code auto-generated from interface
- Server is a struct that implements the interface
- Client uses opDispatch to transparently call remote methods with the ease of local calls
- Input/ref parameters are serialized by client
- Ref/out parameters and return value are serialized by server

D @ Weka

RPC

```
interface IRaftService {
    RequestVoteReply requestVote(RaftId, RequestVoteRequest);


    AppendEntriesReply appendEntries(RaftId, AppendEntriesRequest, InSGData);


    InstallSnapshotReply installSnapshot(RaftId, InstallSnapshotChunkRequest);

    void stepDown(RaftId, NodeId);

    void ping(RaftId);
}
```

D @ Weka

RPC

```
interface IRaftService {
    RequestVoteReply requestVote(RaftId, RequestVoteRequest);

    @notrace @rpcEncryptAllBuffers
    AppendEntriesReply appendEntries(RaftId, AppendEntriesRequest, InSGData);

    @rpcEncryptAllBuffers
    InstallSnapshotReply installSnapshot(RaftId, InstallSnapshotChunkRequest);

    void stepDown(RaftId, NodeId);

    void ping(RaftId);
}
```

D @ Weka

Upgrading

- Customers want new features
- Weka operates at huge scales, sometimes thousands of servers in a cluster, 100Ks of clients
- Disruption to workload during upgrade is unacceptable
- The only practical way to upgrade at scale is rolling/incremental upgrade

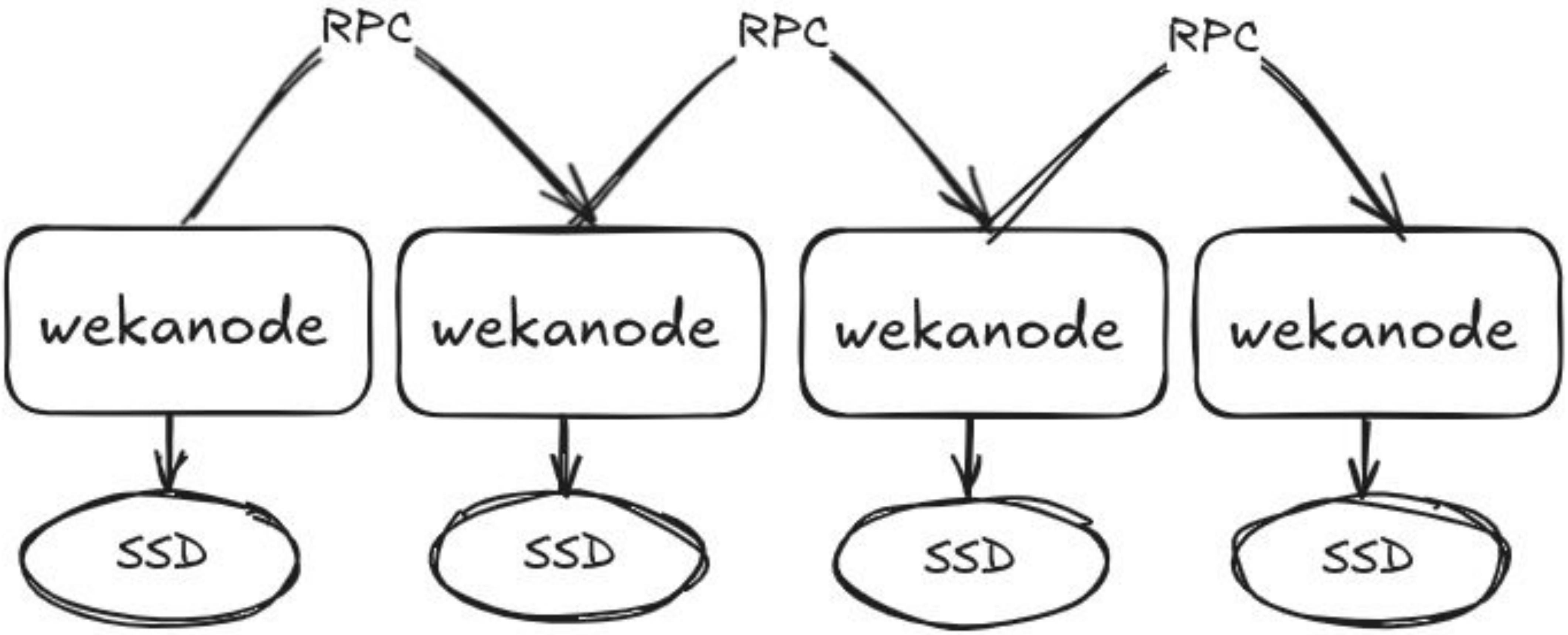
D @ Weka

Upgrading non-disruptively

- Customers want new features
- Weka operates at huge scales, sometimes thousands of servers in a cluster, 100Ks of clients
- Disruption to workload during upgrade is unacceptable
- The only practical way to upgrade at scale is rolling/incremental upgrade

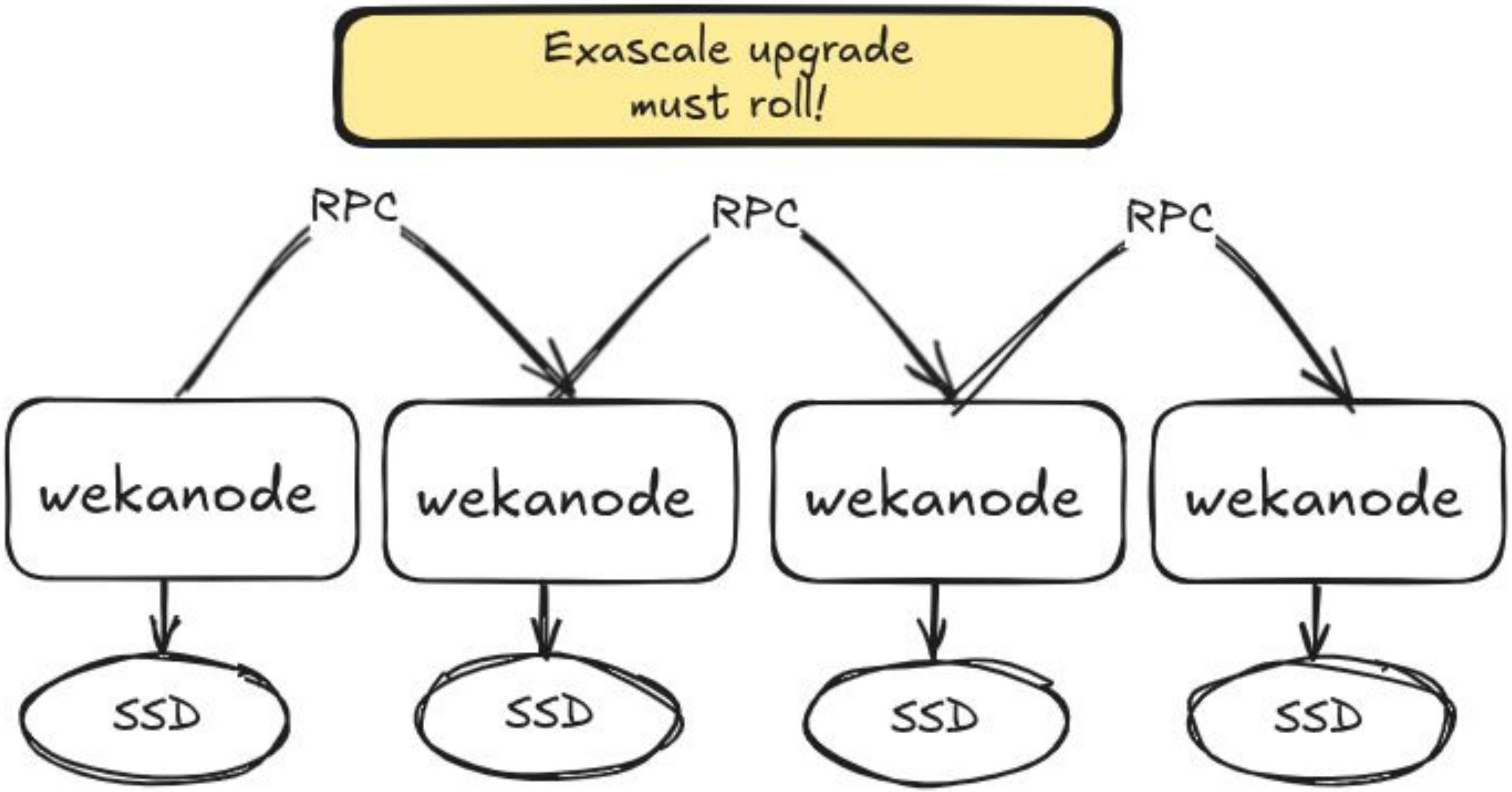
D @ Weka

Upgrading non-disruptively



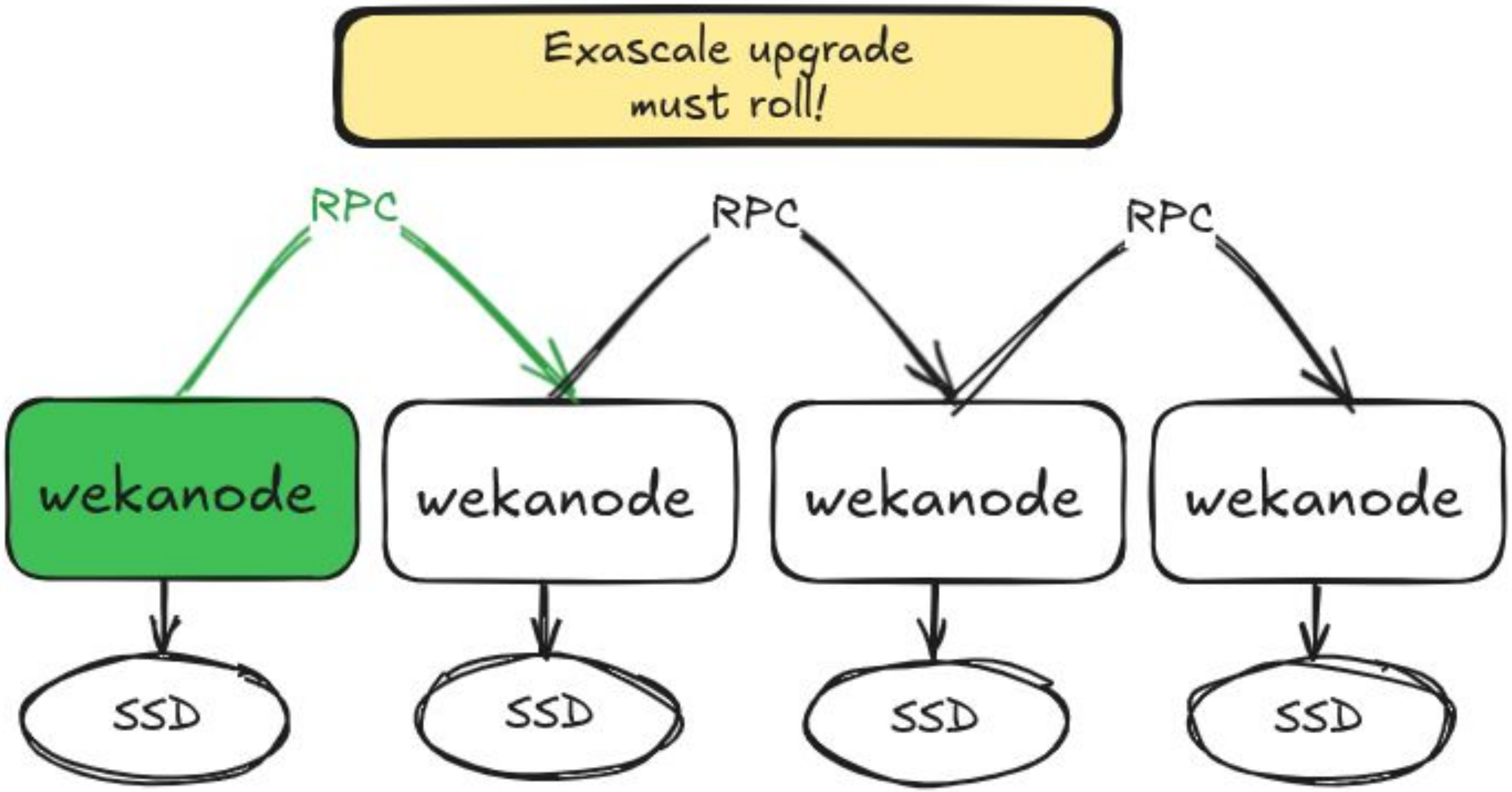
D @ Weka

Upgrading non-disruptively



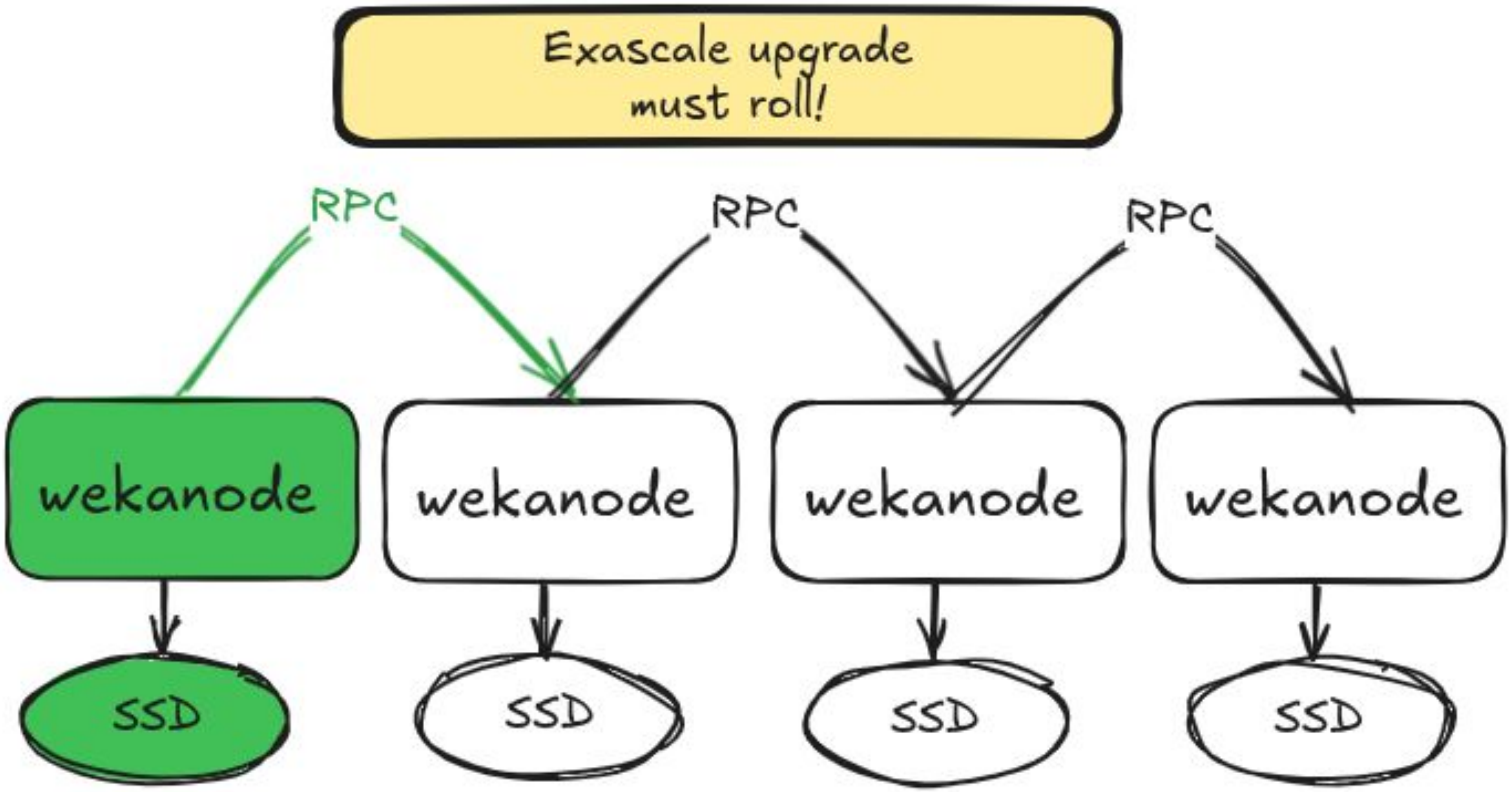
D @ Weka

Upgrading non-disruptively



D @ Weka

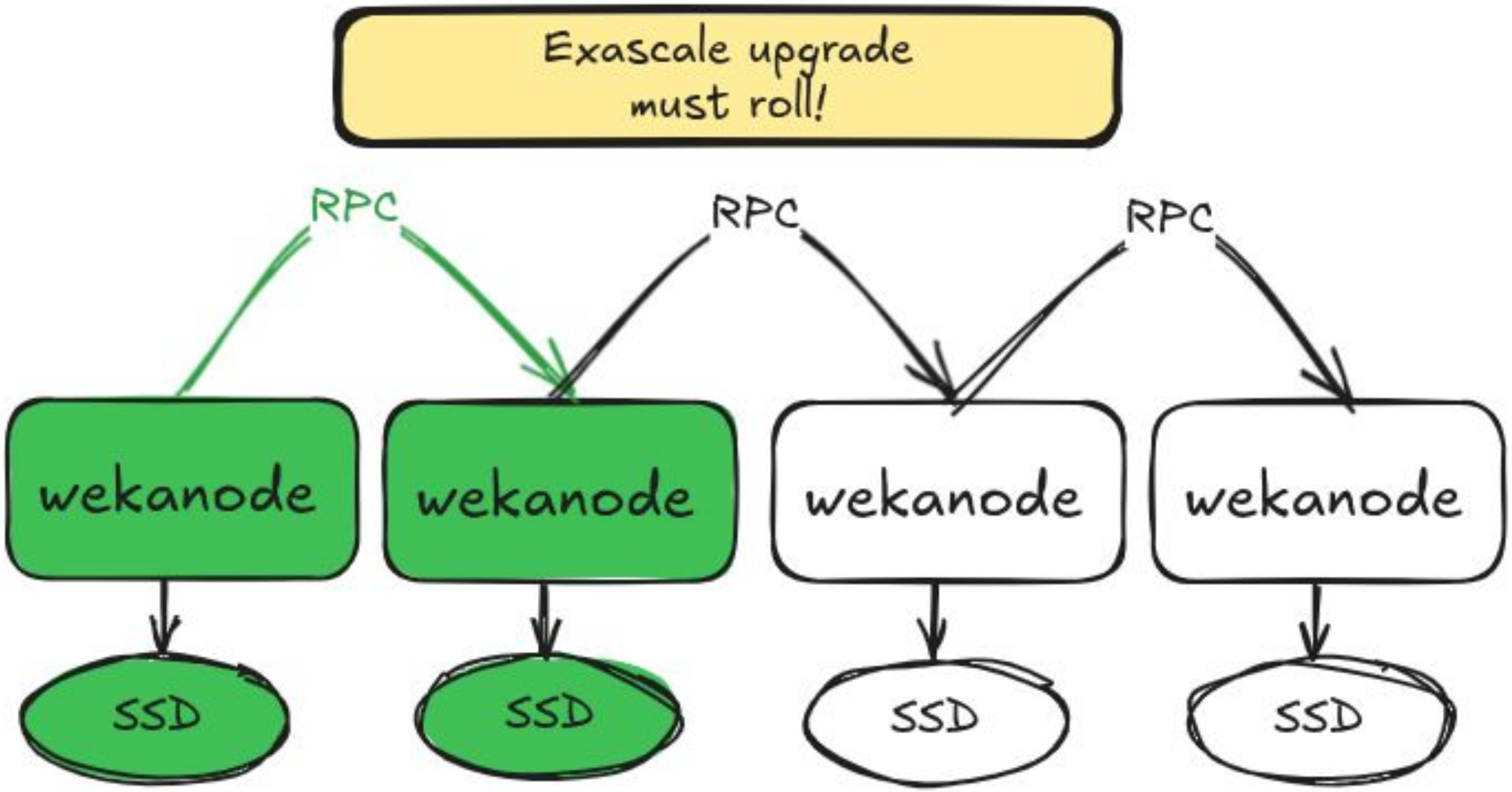
Upgrading non-disruptively



WEKA

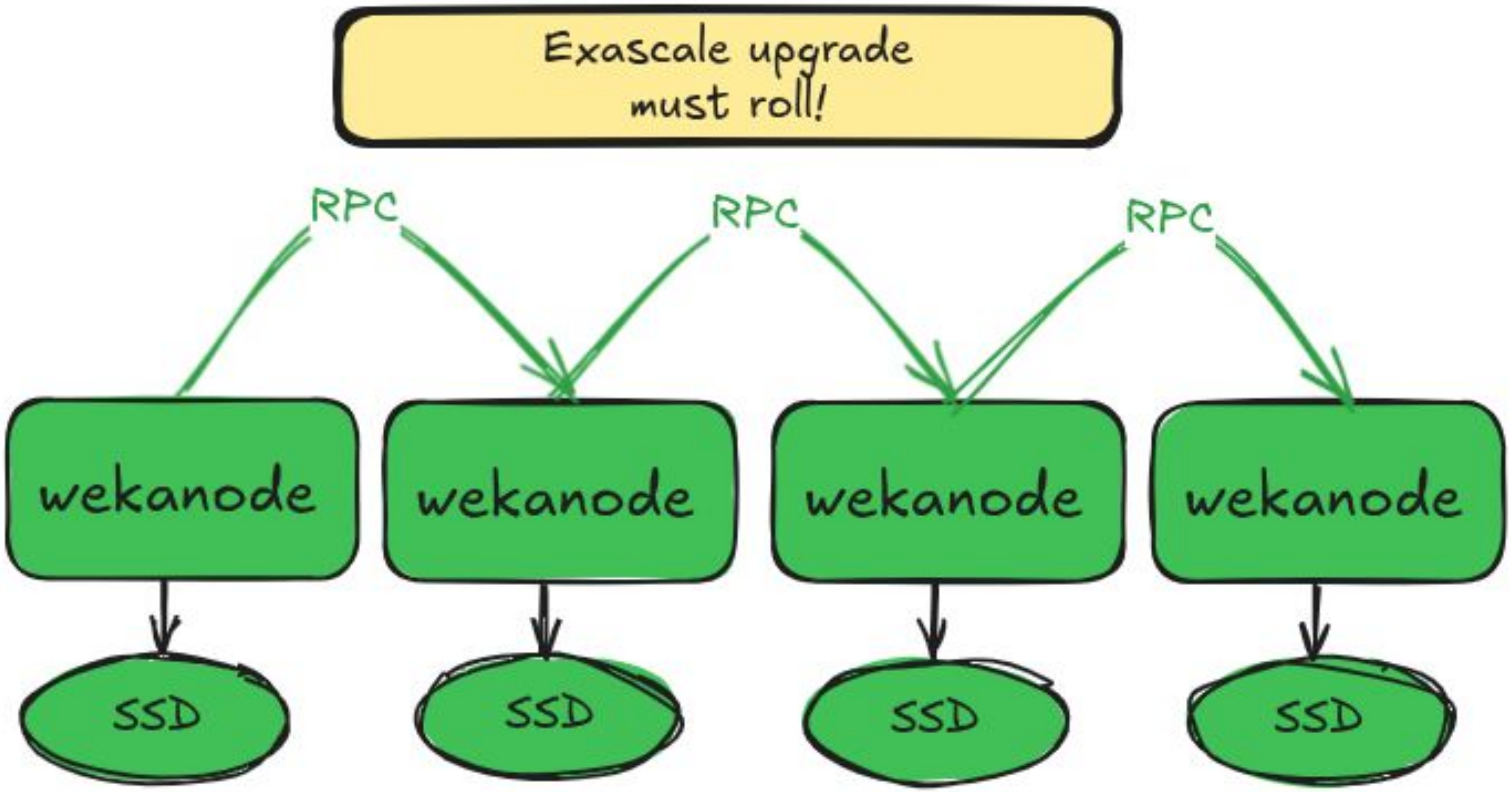
D @ Weka

Upgrading non-disruptively



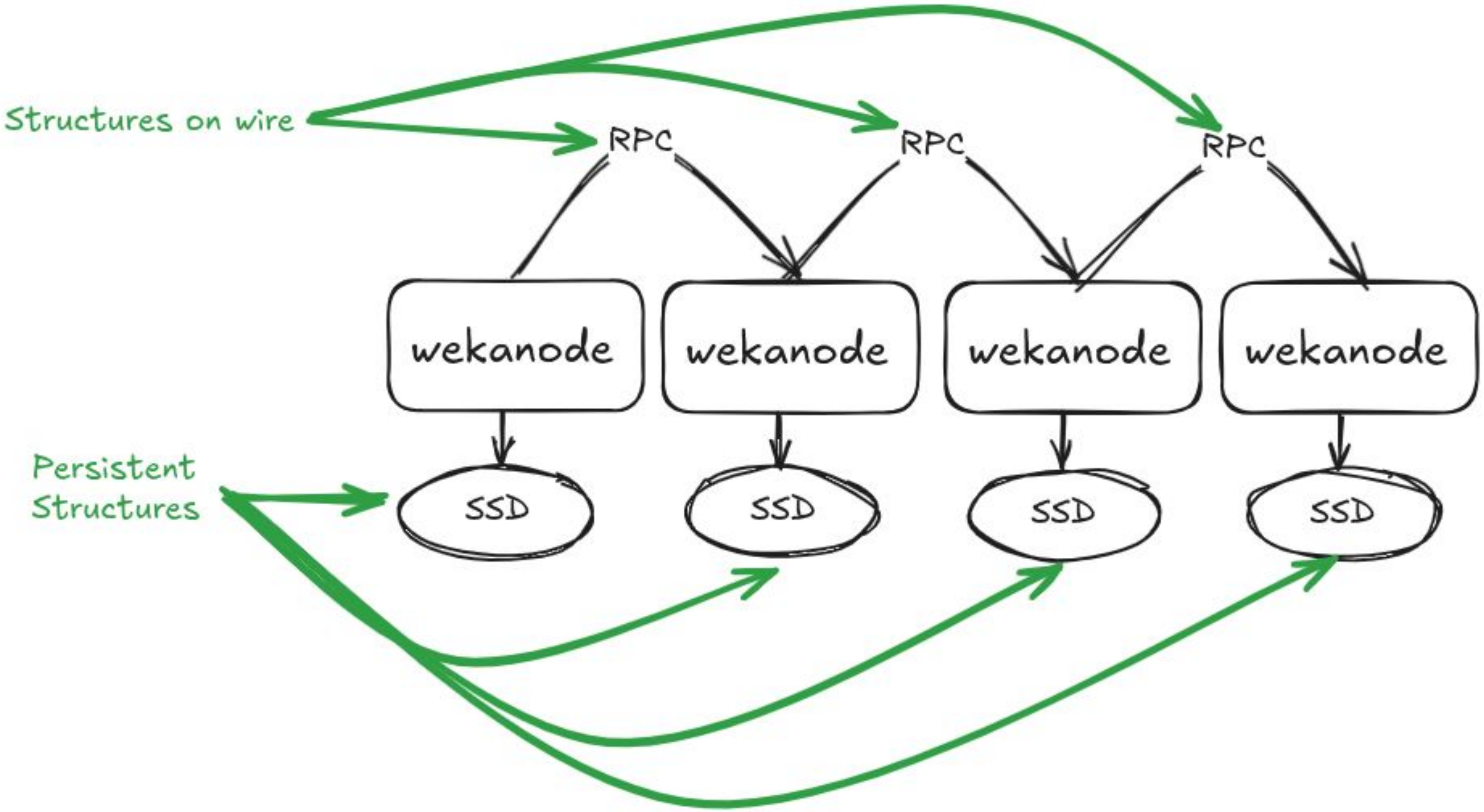
D @ Weka

Upgrading non-disruptively



D @ Weka

Upgrading non-disruptively



D @ Weka

Upgrading non-disruptively

- We modify our structures all the time!
- How can we retain compatibility across versions?
- Upgrade persistent data structures
- Upgrade RPC calls from old RPC clients

D @ Weka

Upgrading non-disruptively

- We modify our structures all the time!
- How can we retain compatibility across versions?
- Upgrade persistent data structures
- Upgrade RPC calls from old RPC clients
- **Downgrade** RPC calls **to** old RPC servers

D @ Weka

Upgrading non-disruptively

- We modify our structures all the time!
- How can we retain compatibility across versions?
- Upgrade persistent data structures
- Upgrade RPC calls from old RPC clients
- **Downgrade** RPC calls **to** old RPC servers

- How does Weka achieve this?

D @ Weka

Ref-types as schemas

- D metaprogramming to the rescue!
- Recursively iterate **all** persistent & RPC types
- Dump their descriptions to a **“ref-types”** JSON file
- Convert these JSON files to .d “old type” and “old interface” declarations
- This is essentially the “schema” **of a specific version**
- Ref-types for each supported source version in repository
- Upgrade/downgrade from/to an old version remains high-performance binary

D @ Weka

Automatic Upgrade/Downgrade

- RPC server should upgrade all inputs and downgrade all outputs
- RPC client should downgrade all inputs and upgrade all outputs
- Persistent data structures should be upgraded

D @ Weka

Automatic Upgrade/Downgrade

- We want to automate as much of this as possible

D @ Weka

Automatic Upgrade/Downgrade

- We want to automate as much of this as possible
- Recursively compare old & current data types
  - Any modifications must be accompanied by UDAs (`@newField`, `@removedField`, ...)

D @ Weka

Automatic Upgrade/Downgrade

- **We want to automate as much of this as possible**
- **Recursively compare old & current data types**
  - Any modifications must be accompanied by UDAs (`@newField`, `@removedField`, …)
  - Otherwise, compilation errors arise

D @ Weka

Automatic Upgrade/Downgrade - Example

```
struct File {
    Time atime, ctime, mtime;
    // ...
}
```

D @ Weka

Automatic Upgrade/Downgrade - Example

```
struct File {
    Time atime, ctime, mtime;
    // ...
}
```

- Feature request: Add “birth time” to files

D @ Weka

Automatic Upgrade/Downgrade - Example

```
struct File {
    Time atime, ctime, mtime, btime;
    // ...
}
```

- Feature request: Add “birth time” to files

D @ Weka

Automatic Upgrade/Downgrade - Example

```
struct File {
    Time atime, ctime, mtime, btime;
    // ...
}
```

ERROR:

```
file.d(100): field 'btime' was added. To allow upgrade, use @newField!<ver>("btime") or
file.d(100): add a custom upgrade method.
convert.d(161): Error: static assert:  "cannot upgrade weka.file.File"
```

WEKA

D @ Weka

Automatic Upgrade/Downgrade - Example

```
@newField!V4_4_11(“btime”)
struct File {
    Time atime, ctime, mtime, btime;
    // ...
}
```

D @ Weka

Automatic Upgrade/Downgrade - Example

```
@newField!V4_4_11(“btime”)
struct File {
    Time atime, ctime, mtime, btime;
    // ...
}
```

✅

D @ Weka

Automatic Upgrade/Downgrade - Example

- Product says: “btime” should not be zero on upgrade!
- Use minimum of existing times as an approximation

D @ Weka

Automatic Upgrade/Downgrade - Example

```
struct File {
    Time atime, ctime, mtime, btime;



    // ...
}
```

D @ Weka

Automatic Upgrade/Downgrade - Example

```
struct File {
    Time atime, ctime, mtime, btime;
    void upgradeFieldFrom(string field: “btime”, Old)(ref const(Old) old) {
        this.btime = min(old.atime, old.ctime, old.mtime);
    }
    // ...
}
```

D @ Weka

Upgrade Summary

- Weka has successfully non-disruptively upgraded customers thousands of times
- In practice clusters may mix more than just 2 versions
- Seamless binary upgrade/downgrade

D @ Weka

Custom Compilation Errors

```
struct Location {
    string filename;
    uint line;
    uint column;


}
```

D @ Weka

Custom Compilation Errors

```
struct Location {
    string filename;
    uint line;
    uint column;
    static auto of(alias Decl)() { return Location(__traits(getLocation, Decl)); }

}
```

D @ Weka

Custom Compilation Errors

```
struct Location {
    string filename;
    uint line;
    uint column;
    static auto of(alias Decl)() { return Location(__traits(getLocation, Decl)); }
    string message(string msg) { return format!"%s(%s): %s"(filename, line, msg); }
}
```

D @ Weka

Custom Compilation Errors

```
struct Location {
    string filename;
    uint line;
    uint column;
    static auto of(alias Decl)() { return Location(__traits(getLocation, Decl)); }
    string message(string msg) { return format!"%s(%s): %s"(filename, line, msg); }
}

struct Struct {
    int field;
}

enum Location location = Location.of!(Struct.field);
pragma(msg, location.message("HI!"));
```

D @ Weka

Custom Compilation Errors

```
struct Location {
    string filename;
    uint line;
    uint column;
    static auto of(alias Decl)() { return Location(__traits(getLocation, Decl)); }
    string message(string msg) { return format!"%s(%s): %s"(filename, line, msg); }
}

struct Struct {
    int field;   // testlocation.d(12): HI!
}

enum Location location = Location.of!(Struct.field);
pragma(msg, location.message("HI!"));

$ dmd location.d
testlocation.d(12): HI!
```

CTFE UT

Unit Tests

- Weka is a **BIG** software project
- Some modules have quite heavy subsystem dependencies
- Compiling and linking Unit Test executables for large subsystems can take a long while!
- This is painful when developing and running UTs

CTFE UT

Unit Tests

- Weka is a **BIG** software project
- Some modules have quite heavy subsystem dependencies
- Compiling and linking Unit Test executables for large subsystems can take a long while!
- This is painful when developing and running UTs

```
auto ut(alias F)() {
    bool check() {
        F();
        return true;
    }
    assert(check());
    static assert(check());
}
```

CTFE UT

Unit Tests

- Weka is a **BIG** software project
- Some modules have quite heavy subsystem dependencies
- Compiling and linking Unit Test executables for large subsystems can take a long while!
- This is painful when developing and running UTs

```
auto ut(alias F)() {
    bool check() {
        F();
        return true;
    }
    assert(check());            // Execute the `F` function in runtime
    static assert(check());
}
```

CTFE UT

Unit Tests

- Weka is a **BIG** software project
- Some modules have quite heavy subsystem dependencies
- Compiling and linking Unit Test executables for large subsystems can take a long while!
- This is painful when developing and running UTs

```
auto ut(alias F)() {
    bool check() {
        F();
        return true;
    }
    assert(check());            // Execute the `F` function in runtime
    static assert(check());     // Execute the `F` function in CTFE
}
```

CTFE UT

Unit Tests - Example

```
int fib(int idx) {
    auto cur = 0;
    auto next = 1;
    foreach(i; 0..idx) {
        auto add = cur + next;
        cur = next;
        next = add;
    }
    return cur;
}
```

CTFE UT

Unit Tests - Example

```
unittest {

        assert(fib(0) == 0);
        assert(fib(2) == 1);
        assert(fib(5) == 5);
        assert(fib(-1) == 0);

}
```

CTFE UT

Unit Tests at compile time - Example

```
unittest {
    ut!({ // this UT will now run in CTFE & runtime
        assert(fib(0) == 0);
        assert(fib(2) == 1);
        assert(fib(5) == 5);
        assert(fib(-1) == 0);
    });
}
```

CTFE UT

Unit Tests at compile time - Example

```
unittest {
    ut!({ // this UT will now run in CTFE & runtime
        assert(fib(0) == 0);
        assert(fib(2) == 1);
        assert(fib(5) == 5);
        assert(fib(-1) == 0);
    });
}
```

BONUS: Semantics of CTFE may differ from runtime, this can detect issues.

CTFE UT

Unit Tests at compile time - Example

```
unittest {
    ut!({ // this UT will now run in CTFE & runtime
        assert(fib(0) == 0);
        assert(fib(2) == 1);
        assert(fib(5) == 5);
        assert(fib(-1) == 0);
    });
}
```

```
BONUS: Semantics of CTFE may differ from runtime, this can detect issues.

    if(__ctfe) ..bug..
```

CTFE UT

Unit Tests at compile time - Debugging

- gdb style debugging is not possible in CTFE
- **`pragma(msg, …)`** is only usable for template parameters and values declared as `enum`
- However, print-based debugging is possible with **`core.builtins.__ctfeWrite`**:

```
/// Writes `s` to `stderr` during CTFE (does nothing at runtime).
void __ctfeWrite(scope const(char)[] s) @nogc @safe pure nothrow {}
```

CTFE UT

Unit Tests at compile time - Debugging

- gdb style debugging is not possible in CTFE
- **`pragma(msg, …)`** is only usable for template parameters and values declared as `enum`
- However, print-based debugging is possible with **`core.builtins.__ctfeWrite`**:

```
/// Writes `s` to `stderr` during CTFE (does nothing at runtime).
void __ctfeWrite(scope const(char)[] s) @nogc @safe pure nothrow {}
                  ^^^ may require GC string formatting
```

CTFE UT

Unit Tests at compile time - Debugging

- Wrapper to allow `@nogc nothrow` use:

```
@safe @nogc pure nothrow
auto ctfeWrite(string fmt, Args...)(auto ref Args args) {
    if(!__ctfe) return;

    // what happens in CTFE stays in CTFE...
    string s = as!"@safe @nogc pure nothrow"(() => format!fmt(args));
    __ctfeWrite(s);
}
```

Cheating the compiler

as

```
as!"@safe @nogc pure nothrow"({ … code … });
```

Cheating the compiler

as

```
as!"@safe @nogc pure nothrow"({ … code … });
```

- Needed when incrementally introducing `@nogc` to a large project
- Needed for legitimate edge cases (such as `ctfeWrite`, assert failure case, …)

Typed Identifiers

```
alias DiskId = TypedIdentifier!("DiskId", ushort, ushort.max, ushort.max);
alias InodeId = TypedIdentifier!("InodeId", ulong, 0, 0, FMT("0x{_value!%016x}"));
alias FSId = TypedIdentifier!("FSId", uint, uint.max, uint.max);
…
```

Typed Identifiers

```
alias DiskId = TypedIdentifier!("DiskId", ushort, ushort.max, ushort.max);
alias InodeId = TypedIdentifier!("InodeId", ulong, 0, 0, FMT("0x{_value!%016x}"));
alias FSId = TypedIdentifier!("FSId", uint, uint.max, uint.max);
…
```

- Confused two identifiers? Compiler error
- Great as documentation
- Nicer string formatting
- Can efficiently search all traces about `DiskId<5>` or `InodeId<0x…>`

Manual Overrides

```
auto timeout = manualOverride.getValue!(“cluster.raft_timeout”, 1500.msecs);
```

Easy to use

This use of `getValue` is all that’s needed to declare this override

Listable:

```
$ weka debug override list-keys
```

statically typed:

```
$ weka debug override add --key cluster.raft_timeou --value '“abc”'
error: Override Key cluster.raft_timeou not found
```

```
$ weka debug override add --key cluster.raft_timeout --value '“abc”'
error: value of type Duration expected for this override: Got "abc"
```

Runtime-efficient: compiles to reading a single `__gshared` variable

Thank You!

in @wekaio

▶ /wekaio

@wekaio

WEKA