From my son’s windows laptop to the cloud

By Alexander Vershilov

What you'll hear in this talk

- How to build reliable solutions that can work in a different environments
- Why haskell can make more good than harm on the way to implement it

About myself & Tweag I/O

- Myself
  - Senior software developer
  - Working at Tweag I/O since it’s foundation in 2013.
  - Previously worked at Parallel Scientific and number of Haskell companies

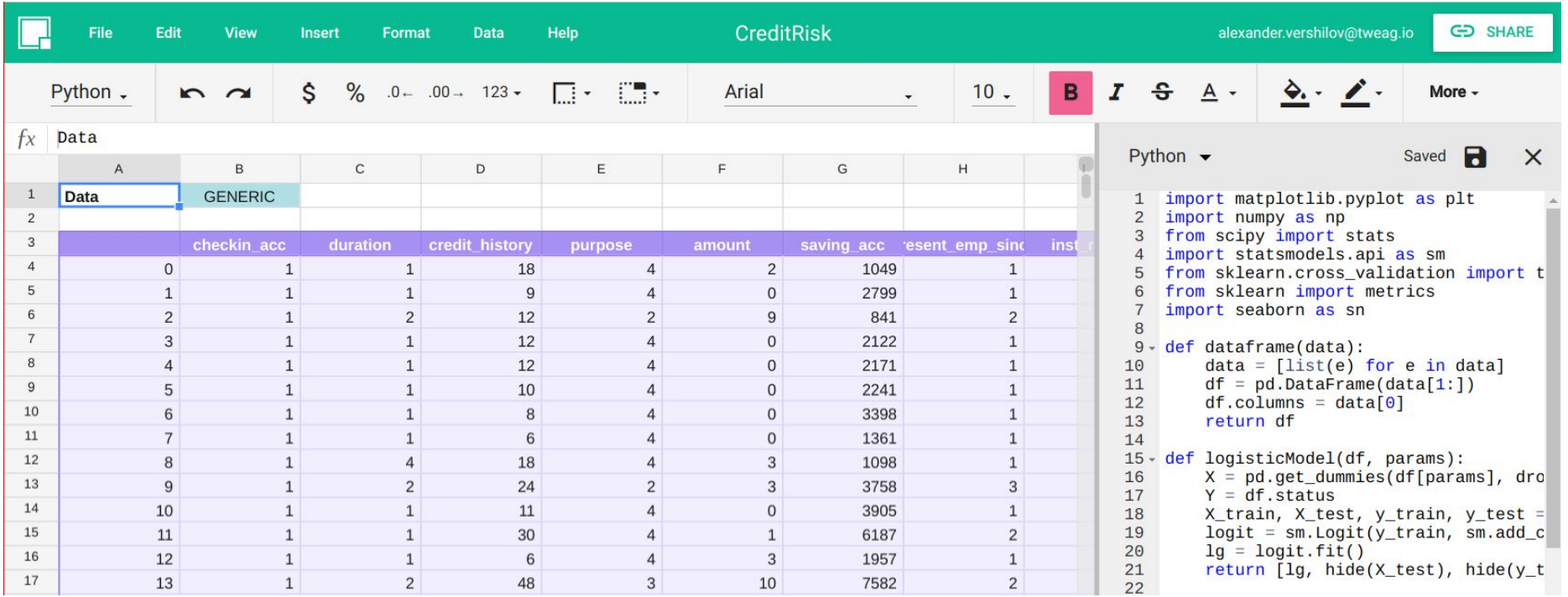
- Tweag I/O
  - Software innovation lab (R&D)
  - Research (linear types and static pointers)
  - Innovation products (SAGE, inline-r, inline-java)
  - Scientific applications (Natural Language Processing, Physics and human body modeling)
  - Haskell and DevOps consultancy

âsheets

https://www.alphasheets.com/

Make Data-Driven Decisions, Faster

AlphaSheets brings the full power of Python, R and SQL to your business data in a familiar spreadsheet interface.



Analyst loves Excel but needs more tools from different languages - so they get it!

Relationship between AlphaSheets and Tweag I/O

Consultancy and POC implementation

Consultancy and implementation

(1 person Tweag)

Code audit and support

Code audit, improving code quality.

(1 person Tweag)

2016 mid summer

2016 end of summer

2016 fall

2017

Work on a component related to inline-r

Consultancy and short contracts

(1 person))

Team augmentation

Architecturing wider system and new functionality, most of the backend and infrastructure development

(3 persons Tweag)

Complexities of the project

- Legacy code
  - It’s possible to write non-ideal haskell code
  - Some “patterns” may not work well but this may only become clear in hindsight
  - There were few large refactors ongoing
- Spreadsheet product - is quite complex domain
- Continuous delivery - we should provide new features without large delays for refactoring
- Frequent requirements update based on the customer experience

Targets that were required

1. Public cloud version
2. Private cloud version
3. Standalone version **(Windows)**
    a. Non-priviledged user
    b. Priviledged user
4. Full on premise version

How we got around all of these requirements

How to write the solution that can scale from 1 node to 1k and work well in both cases?

- Approach 1: Use the most advanced cluster system everywhere (e.g. MiniKube)
  - Require lots of software to install
  - Has a large overhead on a small laptop
- Approach 2: Customize a system for each target reusing best components possible
  - Allow to fit the requirements of each system
  - Harder to test

General approach

- Reuse external libraries and solutions as much as possible
  - Haskell ecosystem and teams are relatively small, so it's nice to use external powers to make things work
    - Google Cloud Platform (compute platform, logs, metrics, scaling, alerts)
    - Kubernetes
    - **Nix**
    - inline-r
    - Jenkins/External CI
- Abstract components and leaving clearly defined protocols between them

What language and solution should provide

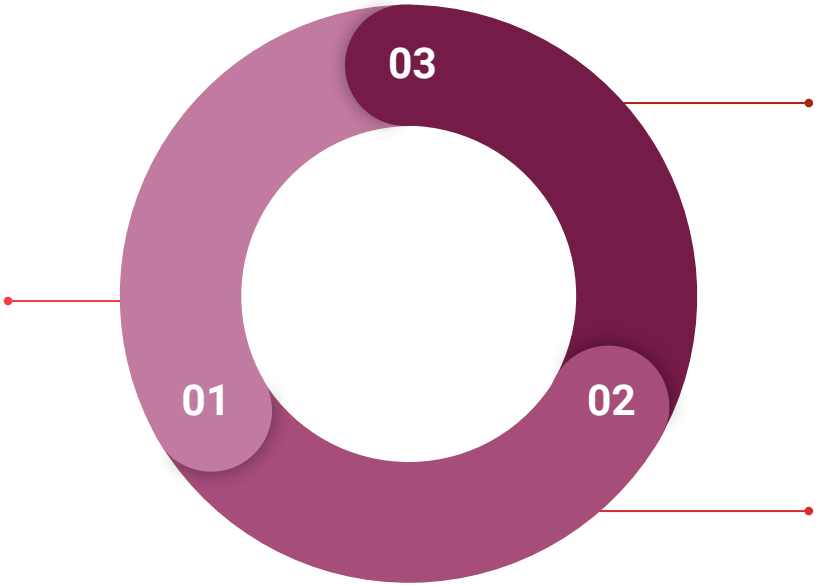
1. Rapid prototyping
2. Strong guarantees about written code
3. Ability to refactor code
4. Ability to reason about code
5. Ability to reuse third-party libraries
6. Working on the different platforms without too much effort
7. Decent performance

Implementation process

For each new feature we want

Proof of Concept Prototype

Write an implementation of the subsystem, that specifies design and basic interface. Strong typing helps a lot on this step helping to remove bad API early



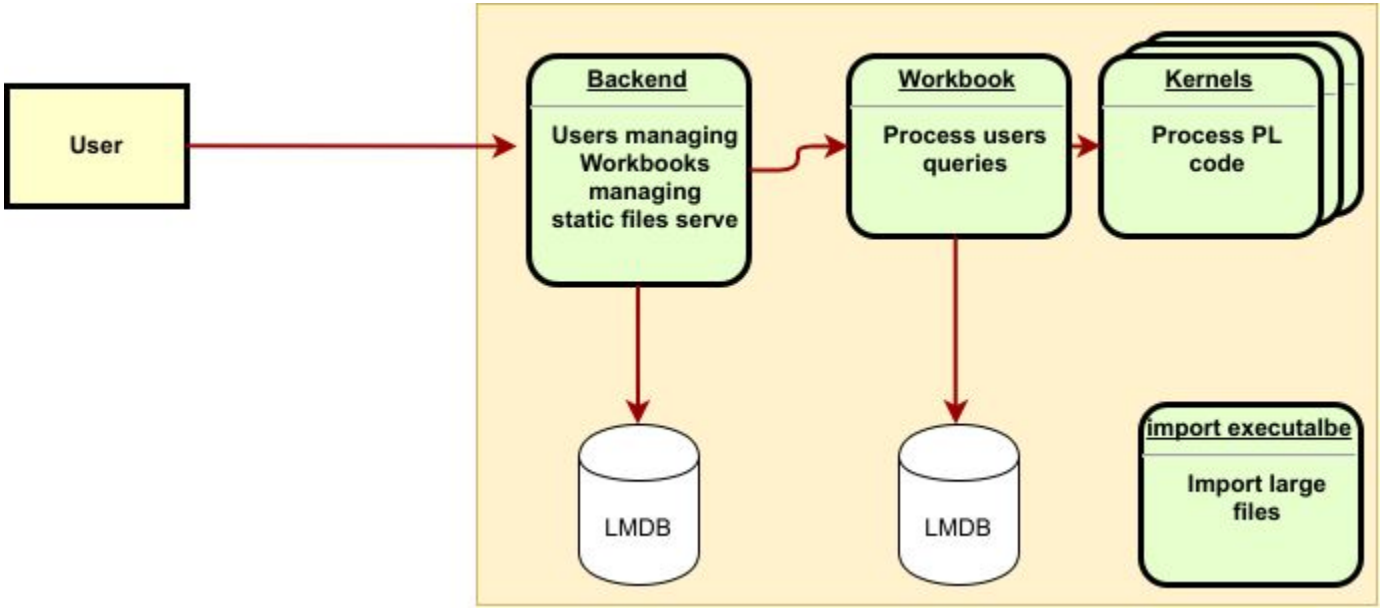
Implement full featured solution

Subsystem can be used by the other subsystems, and they can rely on the properties declared by the subsystem.

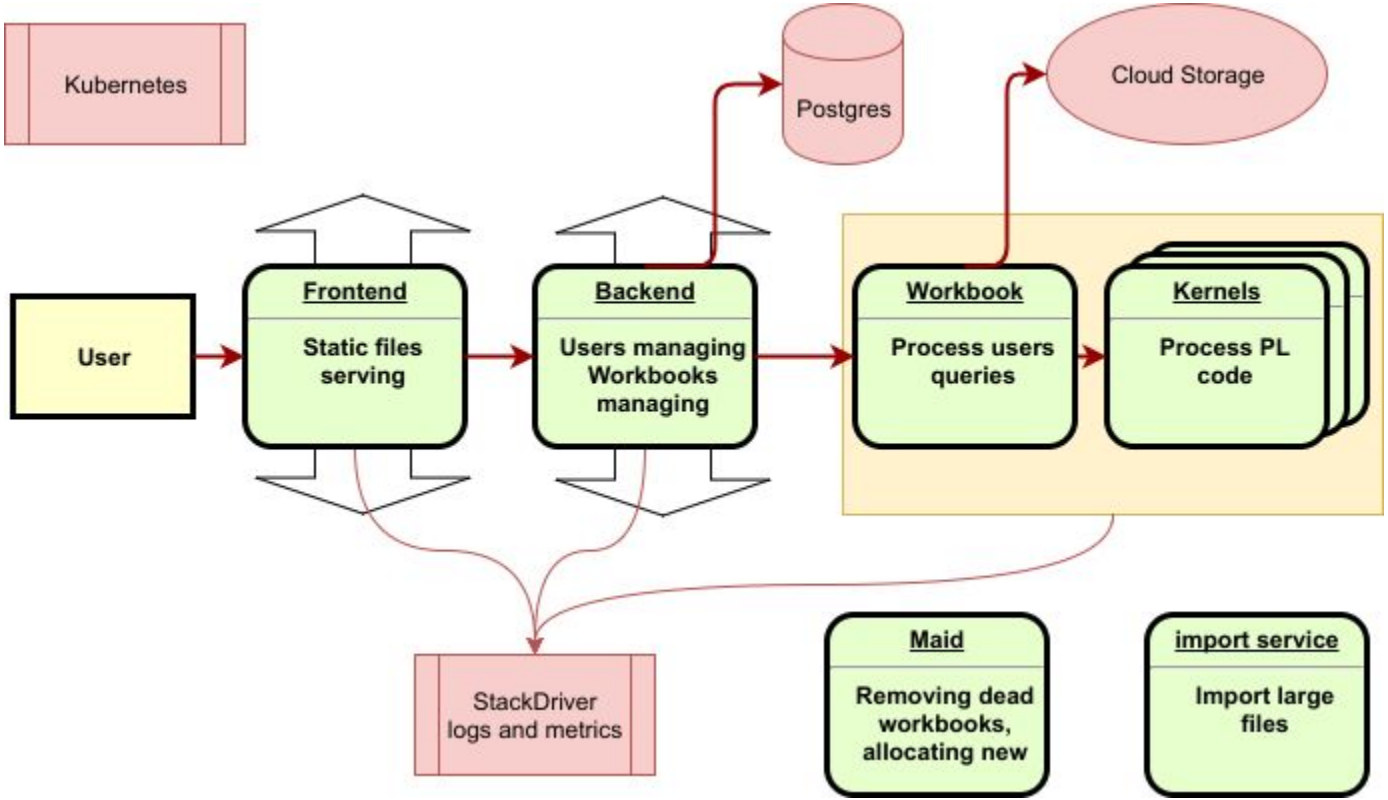
Evaluate if it fits required constraints, implement MVP

Check if implemented solution actually covers requirements, performance, memory wise. Test API sanity.

Standalone version setup



Cluster setup



What abstraction should we use

- Define a clear interfaces and components
- Implement as few code paths as possible
- Each component should provide a strong guarantee about its properties

Services:

- Unit of work, that can be thread or process
- Provide a typed message passing API, that keeps possible effect in types
- Datastructure that is self contained, keeping all locks, message passing inside.

Interfaces:

- Type class interfaces
- Explicitly typed messages and parameters
- Request types that define reply types
- Workbook managing interface

Service pattern: dependency injection

https://www.schoolofhaskell.com/user/meiersi/the-service-pattern popularized by Better

```
module Application.Service.Foo
  ( Service(..)
  , withService
  , ...
  ) where

data Service = Service
  { method1 :: Param1 -> IO Reply1
  , method2 :: Param2 -> IO Reply2
  }
```

```
Module Application.Service.Foo.BarImpl
   ( fooService ) where


fooService :: OtherService IO Service
fooService s = do
   let service = Service
         { method1 = \param1 -> ...
         , method2 = \param2 -> ...
         }
  pure service
```

Example, workbook manager service

```
AS.App.WorkbookManager
data WorkbookManager = WorkbookManager
  { wmGetWorkbookAddress :: WorkbookID -> IO (Either WorkbookServiceException WorkbookInfo)
  , wmRemoveWorkbook :: WorkbookID -> IO (Either WorkbookRemoveError ())
  , …
  }

AS.App.WorkbookManager.Kube
mkManager :: WbWorkbookManagerKubernetesConfig
                -> PersistUserDb
                -> PersistClusterDb
                -> IO (WorkbookManager, IO ())
mkManager = …

AS.App.WorkbookManager.Local
mkManager :: WbWorkbookManagerLocalConfig -> PersistUserDb -> IO (WorkbookManager, IO ())
mkManager = do
  tid <- forkIO $...
  pure $ WorkbookManager $ ...
```

Service pattern: continued

Works really well for splitting interfaces and abstraction boundaries.

Introduces quite low overhead, especially for people who care about compilation time.

Very handy for testing purposes.

Easy to use with conditional compilation

May be hard to understand for users who are not used to this pattern.

Sometimes services have mutual dependencies, so we need `mdo` and friends.

Long dependency chain may be hard to track.

Methods are not specialized even when it’s cheap to do so.

Our approach to implementing a service

- Use services as an internal interface description.
- Use MTL-like monad, i.e. a concrete monad that implements all required interfaces.
- Track features that monad should have in each method - reason about the effects and requirements that each method have.

```
data DbInterface = DbInterface
  { _diEnqueueTransaction :: forall a . DBT 'ReadWrite 'WorkbookDB a -> IO (STM a)
  , _diDelayTransaction :: forall a. DBT 'ReadOnly 'WorkbookDB a -> IO a
  , _diNoBlockTransaction :: forall a. DBT 'ReadOnly 'WorkbookDB a -> IO a
  , getWorkbookDir :: WorkbookDir
  }

class HasDbInterface m where getDbInterface :: m DbInterface
instance Monad m => HasDbInterface (ReaderT DbInterface m) where getDbInterface = ask

withDbInterface
  :: (MonadHasAct m, MonadMask m, MonadAsyncWithUnmask m)
  => PersistedLmdb -> (DbInterface -> m r) -> m r
withDbInterface = …

onWorkbookDbRo :: (HasDbInterface m, MonadIO m) => DBT ‘RealOnly a -> m a
onWorkbookDb :: (HasDbInterface m, MonadIO m) => DBT ‘ReadWrite a -> m a
onWorkbookDbAsync :: (HasDbInterface m, MonadIO m) => DBT ‘ReadWrite a -> m a
```

```
data WorkbookEnv = WorkbookEnv
  { _workbookEnvId :: WorkbookID
  , _workbookDbInterface :: DbInterface
  , _workbookTilesState :: MVar TilesState
  , _workbookSheetsRef :: IORef.IORef [Sheet]
  , _workbookLogger :: !LoggerEnv
  , _workbookTraceState :: !(IORef.IORef Trace.TraceState)
  , _workbookMetry :: !Metrics.Settings
  , _workbookUsersVar :: TVar UsersMap
  , ...
  }

newtype WorkbookM a = WorkbookM { runWorkbookM :: ReaderT WorkbookEnv IO a }

instance HasDbInterface WorkbookM where
  getDbInterface = WorkbookM $ asks _workbookDbInterface
```

Outline

- Haskell type-system allows to build interfaces and track them using types
- In case of refactoring type-system points out all places that doesn’t work

Is it actually enough?

(there was something about Windows in the topic)

Working on windows build package

- Differences with unix:
  - Do not want to pull in PostgresSQL, and kubernetes related libraries everything is stored in LMDB.
  - Use TCP instead of BSD sockets for kernel communication.
  - Do not rely on fork semantics
  - Do not rely on POSIX signals.

Windows: build

- Many ways to set up env. Use stack to install Haskell ecosystem on Windows
- Have stack.windows.yaml that doesn't pull in cluster and db dependencies
- Install system packages using pacman from msys2:
  pacman -S mingw-x86_64-zeromq \
  mingw-w64_x86_64-lmdb \
  -noconfirm -noprogressbar
- Install relocatable version of python
  install packages using using `pip` and just copy python directory to the target
- Wrap executables in haskell one, in case if it requires special dealing with environment and arguments

Working on windows: packaging up

- Build package
  - stack install
  - package shared objects
  - put to the dist folder
- Using shell scripts to gather executables and dependencies in

  `dist`

  `dist/backend`

  `dist/pykernel`

  `dist/static`
- Package everything using NSIS

```
$dll_files = gc ‘haskell-deps.txt’

&’stack’ @(‘install’, ‘--stack-yaml’,’stack.windows.yaml’)

New-Item -Type directory -Name dist\backend -Force
Copy-Item “${env:APPDATA}\local\bin\alphasheets-exe.exe”
dist\backend
Copy-Item “${env:APPDATA}\local\bin\workbook-exe.exe” dist\backend
Copy-Item “${env:APPDATA}\local\bin\pykernel.exe” dist\backend
$dll_path = stack path --extra-library-dirs
foreach ($file in $dll_files) {
  Copy-Item “$dll_path.Split(“,”)[0]” “dist\backend\$file”
}
Copy-Item Environment.cfg dist\
```

Windows: problems

- Need to patch libraries because they are not well tested on windows
  Though changes are rather trivial in most of the cases.
- Haskell runtime is much slower on windows, but it was not a problem in all tested cases
- Latest stack and haskell libraries do not work with text-icu from mingw
- Binary version of mingw is compiled against MSVC runtime library
- Source version has problems with building text-icu library

Current solution: use limited functionality without requiring text-icu.

Future solution: build text-icu in mingw and use it in product.

Do we use same ad-hoc solutions to building software everywhere?

Building software on other systems

- Use nix as a software management tool
- Can work on any linux platform
- Can be used to create developer environments
- Used to build software on CI and run tests

Nix.. Nix.. Nix..

Nix is a generic and composable build system

Use nixpkgs to re-use thousands of human-hours in packaging software

Use Nix to build Haskell, Python, R, JavaScript, ...

Compose the Binaries into Docker containers, only ship the runtime dependencies

For the standalone edition, compose the Docker image into a windows installer

Developers can use the same files to build their own environments on any OS

Developer environment

Developer can run nix-shell where all tools and system packages are built:

- stack, GHC, Python and packages, R with packages, yarn

Use stack to build package inside env.

Developer can use any distributive of his choice and any tools of his choice, while getting exactly the same environment as in production.

Build pipeline

Use stackage2nix for Nix expression generation

nix-build

=> builds everything (don’t use stack there)

=> runs tests

Use buildkite to do impure things like deployments

Testing and pipelines

- Testing is really needed.

- Tests we use:
  - Unit tests
  - Migrations tests
    - Migrations are run on a real databases with a list of simple tests after
  - Integration tests
    - Haskell based client
    - Javascript based client
    - Integration tests are run toward a copy of the real cluster

Recap

Lessons learned

- Finding the right abstractions and interfaces is the key point to writing good and maintainable system
- Haskell helps to build sound interfaces do that by having explicit and powerful type system, it's type-system is strict enough help finding breaking pieces refactoring and find errors earlier, but not too strict to discourage project wide refactorings
- Nix couples well with haskell and allow to build reliable systems and developer environment in a single place

tweag.io

alexander.vershilov@tweag.io

www.tweag.io

More fun with the project:

- Logging system
- Metrics gathering
- Profiling
- Writing efficient DB access
- Internal communication

Logs

- Provides a logging interface:
  - Add log with relevant severity level
  - Add internal context to logs, JSON key-pairs
- Use structured logs that keeps current context and metadata
- Each subsystem and action is adding relevant context
- Use katip for monitoring logs, with modified dump function
- Use StackDriver to observe logs

Metrics and profiling

- Use Logger interface to attach profiling API
- Use eventlogs to gather performance statistics without big performance impact
- Provide logger like context that adds information about the function
- As a result structured view of the time spent in each function
- For memory allocations fallback to standard haskell profiling mechanisms
- All data during tests run is sent to Kibana

Databases

Use PostgreSQL for the users and database

Use LMDB for workbook database and user database in standalone version

Almost totally rewritten lmdb API library, that provides zero-copy interface. State of the existing LMDB libraries is quite poor. You are open to get additional copies, segfaults and problems of any kind.

Use streaming library in order to provide safe access to the database memory without additional data copying.

Communication

Websockets - for user to workbook communication.

Zeromq - for internal communication.

State of the library, how it works

Tried store, binary and cbor - in the end stopped on store as it fits our model best of all and we don’t communicate with libraries written in other languages.