# Pipeline Launcher

v0.9.73

Dataflow architecture

TPL.Dataflow

PipelineLauncher demo

# Dataflow architecture. Overview.

**Dataflow architecture** is a computer architecture that directly contrasts the control flow architecture. Dataflow architectures do not have a program counter (in concept): the executability and execution of instructions is solely determined based on the availability of input arguments to the instructions, so that the order of instruction execution is unpredictable, i.e. behavior is nondeterministic.

**Dataflow** is a software paradigm based on the idea of disconnecting computational actors into stages (pipelines) that can execute concurrently. **Dataflow** can also be called stream processing or reactive programming.

# System.Threading.Tasks.Dataflow. Basics.

**TPL Dataflow** is a powerful library that allows you to create a mesh or pipeline and then (asynchronously) send your data through it. Dataflow is a very declarative style of coding; normally, you completely define the mesh first and then start processing data. The mesh ends up being a structure through which your data flows. This requires you to think about your application a bit differently, but once you make that leap, Dataflow becomes a natural fit for many scenarios.

Each mesh is comprised of various blocks that are linked to each other. The individual blocks are simple and are responsible for a single step in the data processing. When a block finishes working on its data, it will pass it along to any linked blocks.

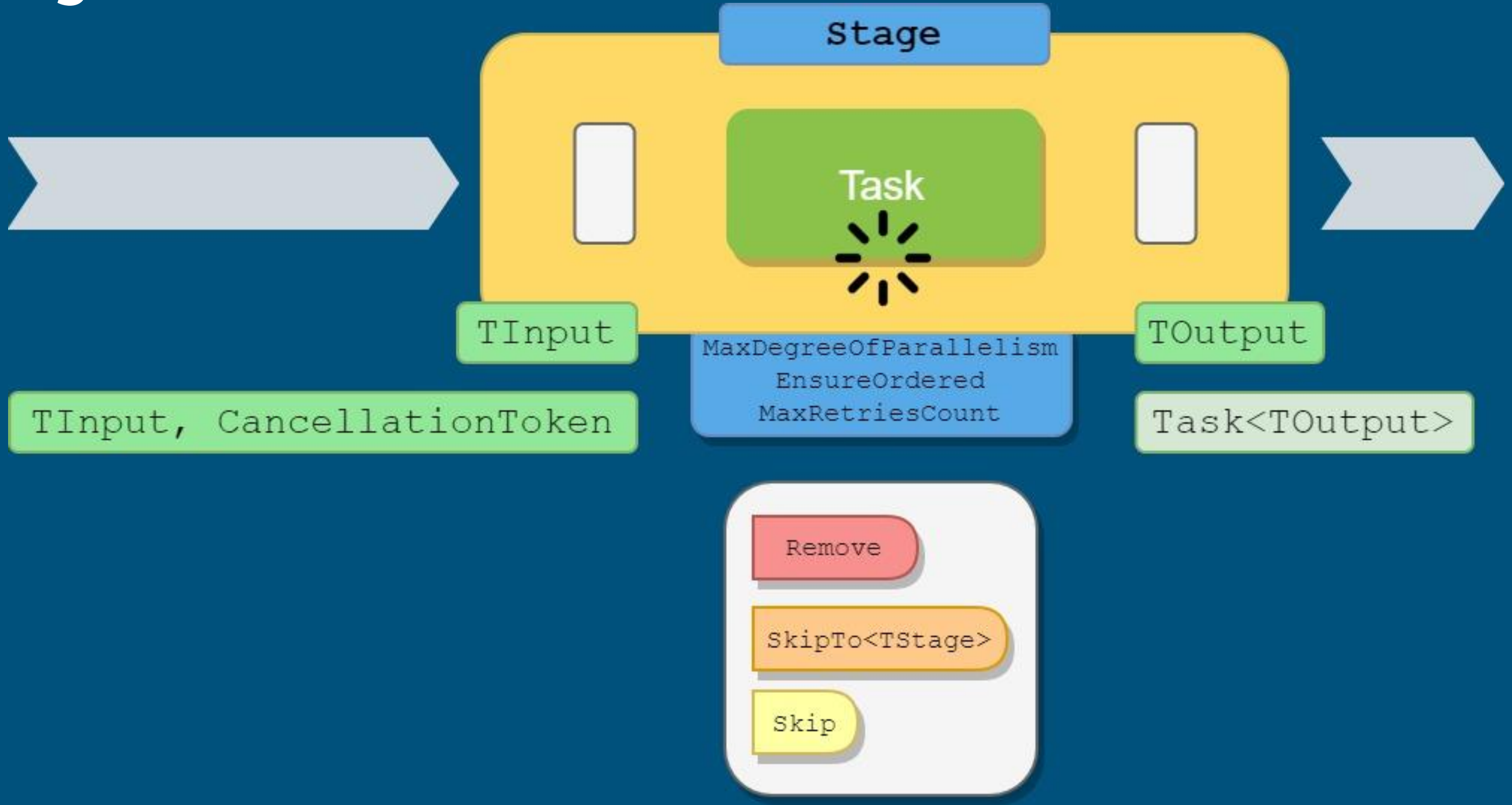
# Pipeline Implementations. Basic types.

1. **Synchronous execution** – In its most simple form, the pipeline object will synchronously execute all of its steps and return the result. Basically, a regular method call.
2. **Asynchronous execution** (producer/consumer) – We might want to have our pipe executions work in some background thread, which we can add jobs to from other threads. This is a subset of the producer/consumer pattern (Job Queues) where each job is a pipeline. This means that the entire pipeline steps will be executed in a single thread for a specific item.
3. **Multi-Threaded pipeline** – With this approach, each step of the pipeline is executed on a separate thread (or threads). There's a buffer (Queue) between each pipeline element to store step results while the next element is still not able to receive them. The buffers might be limited in size.
4. **Multi-Process** pipeline and **Multi-Machine** pipeline – Much like Multithreaded pipeline, we can have pipeline steps across multiple processes or even machines. This might be useful for performance or security measures. For example, just one machine (or process) can access a specific resource (like a database).

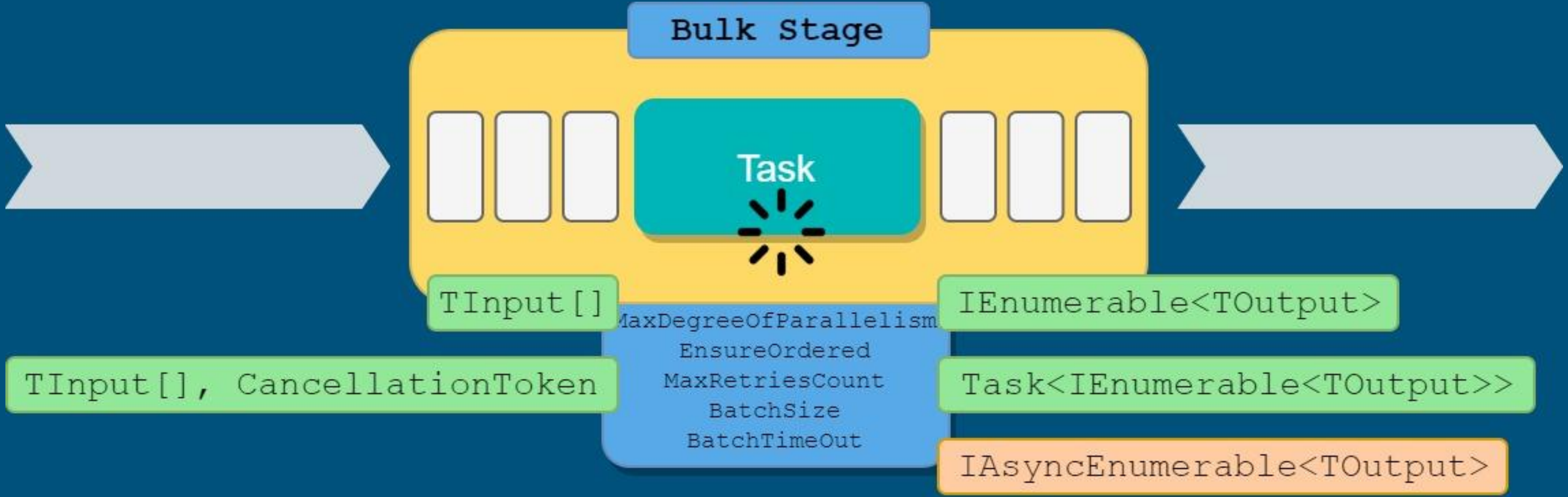
## PipelineLauncher. What is it & what is it for?

- Improve the performance and stability of highly loaded system.
- Is an implementation of the actors model.
- Approach is based on creating a mesh or pipeline, consisting of computing units operating in parallel and independently of each other
- Interaction occurs by transferring asynchronous messages between blocks
- Creating a transparently connected chain of stages attached to each other at the compilation time.

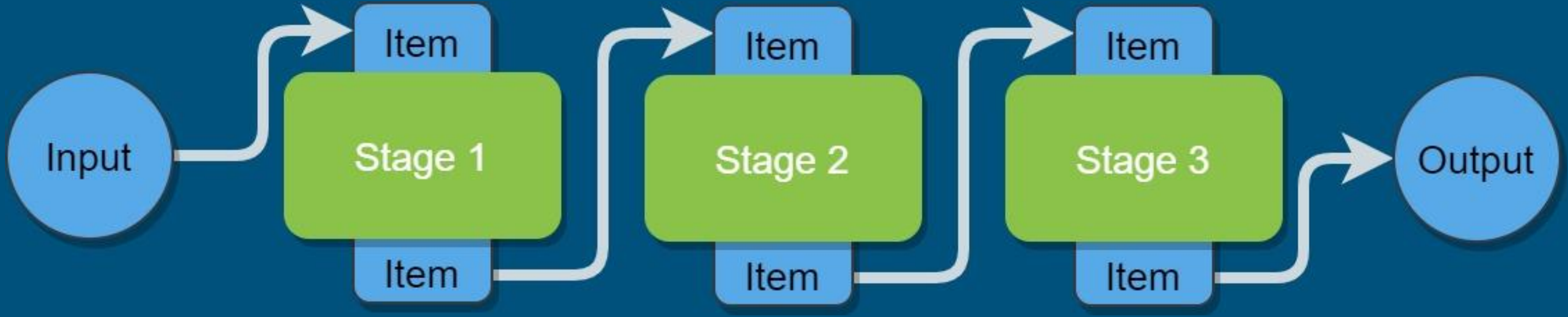
# Stage. Overview.



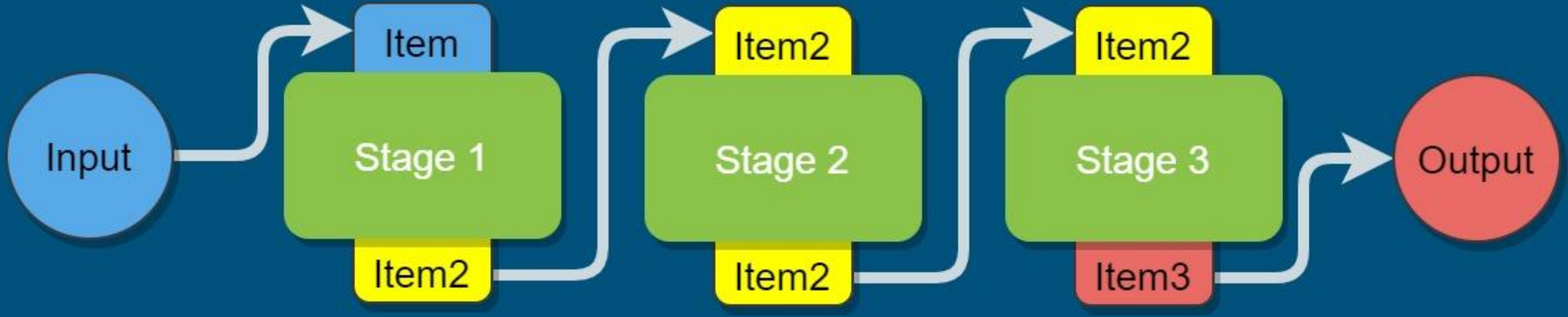
## BulkStage. Overview.



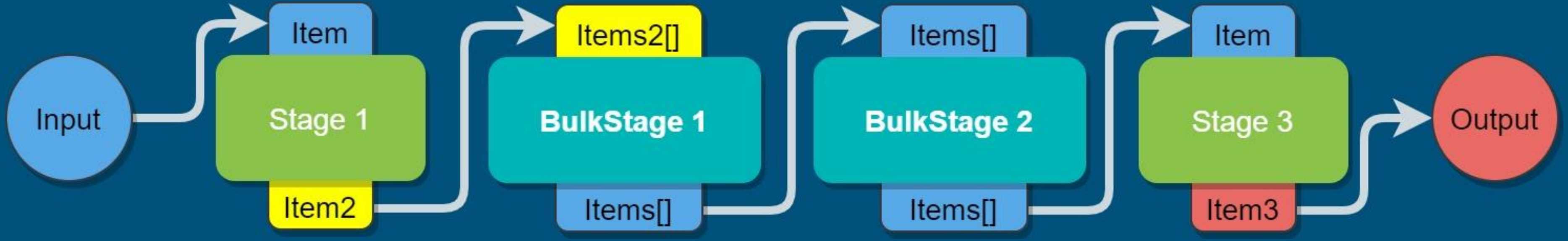
## Pipeline. Simple stages.



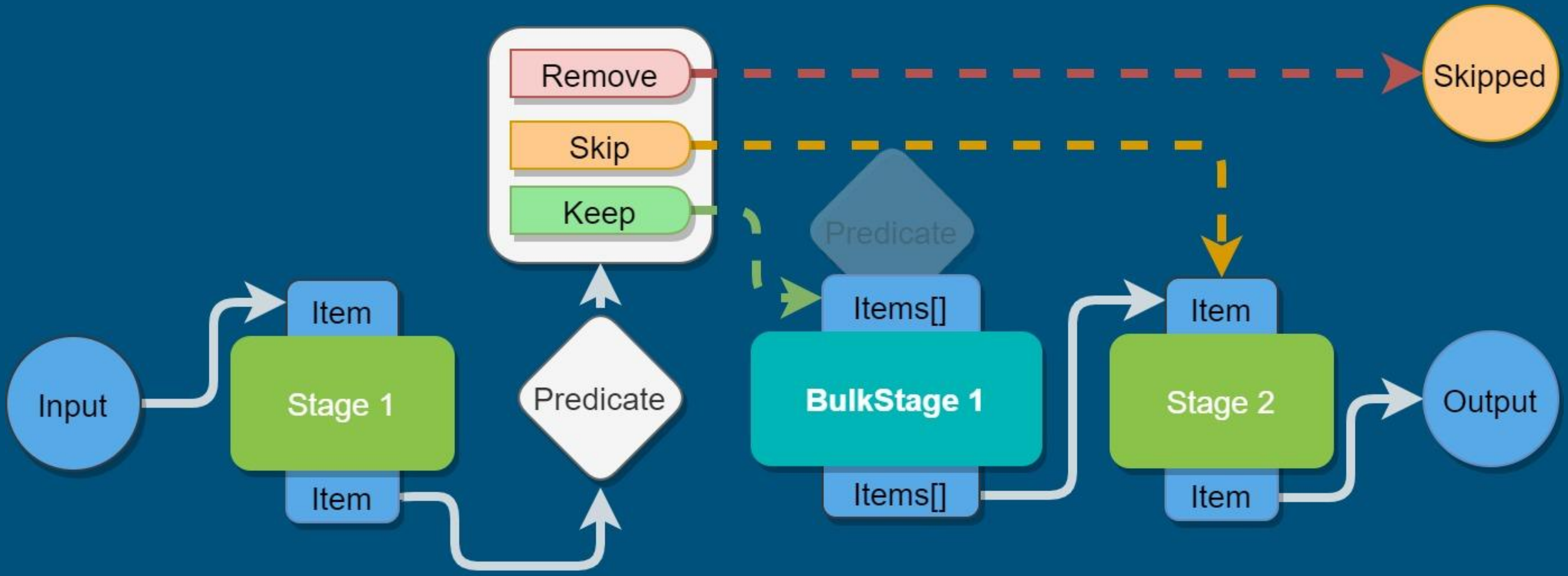
## Pipeline. Item type changing.



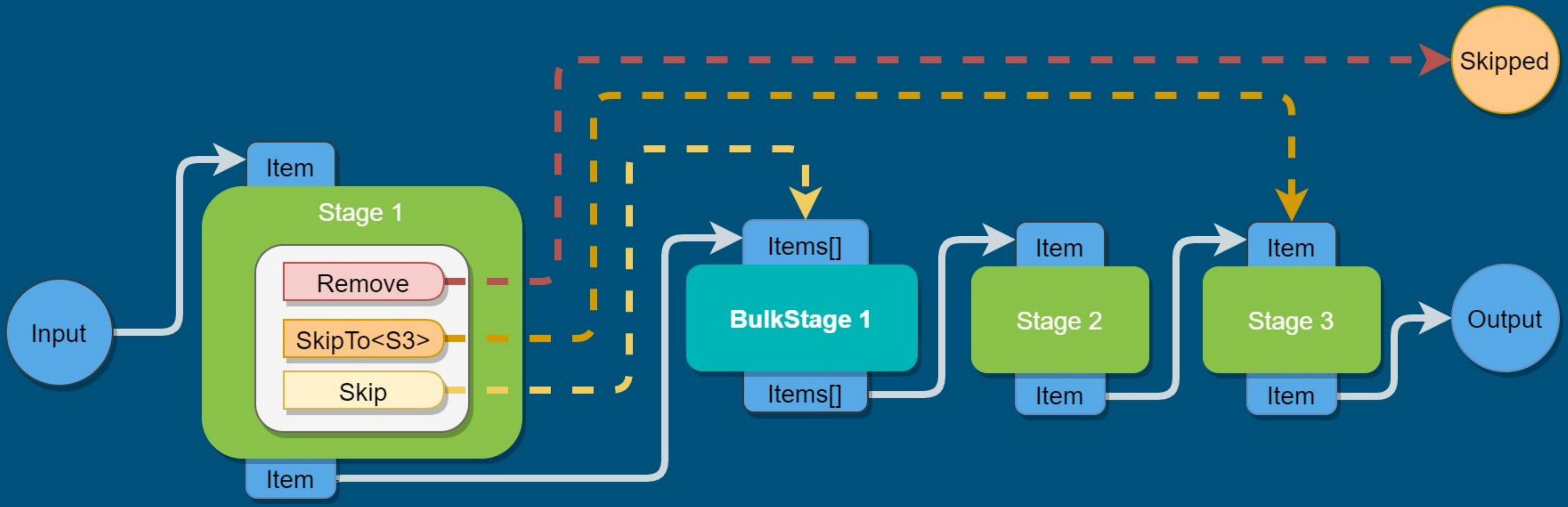
# Pipeline. Bulk stages.



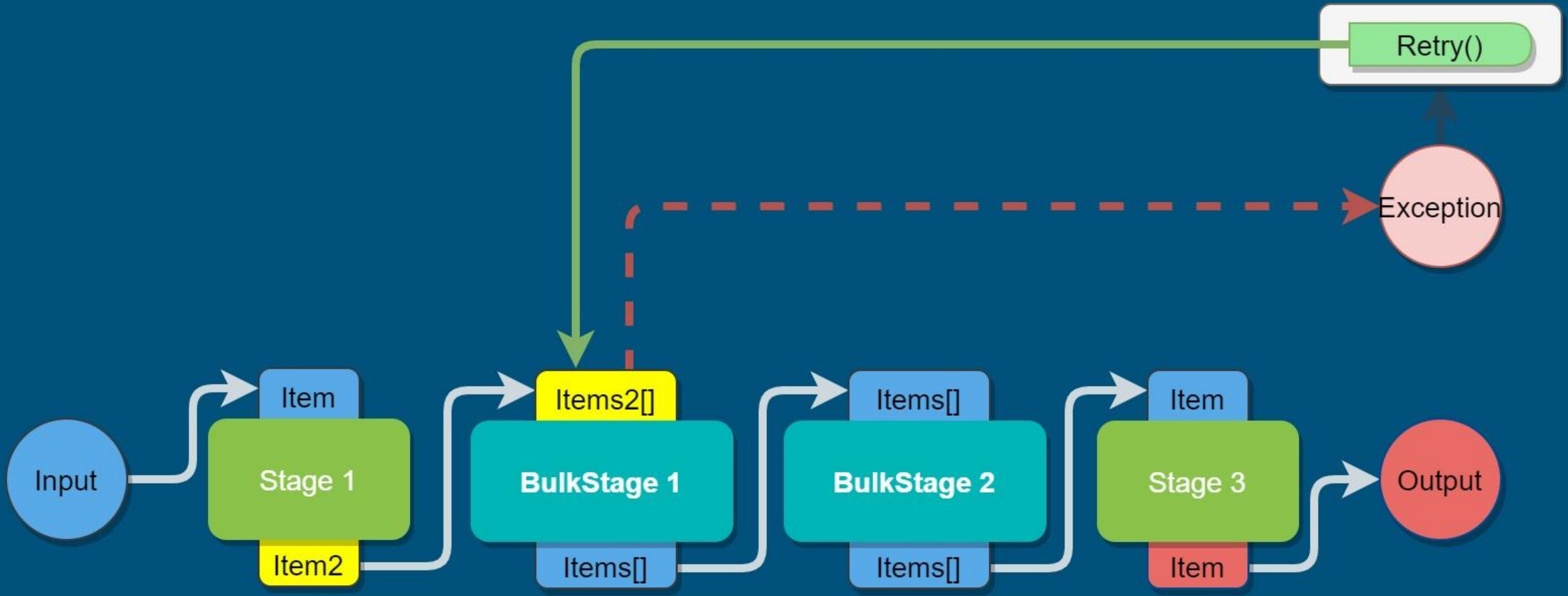
# Pipeline. Predicates between stages.



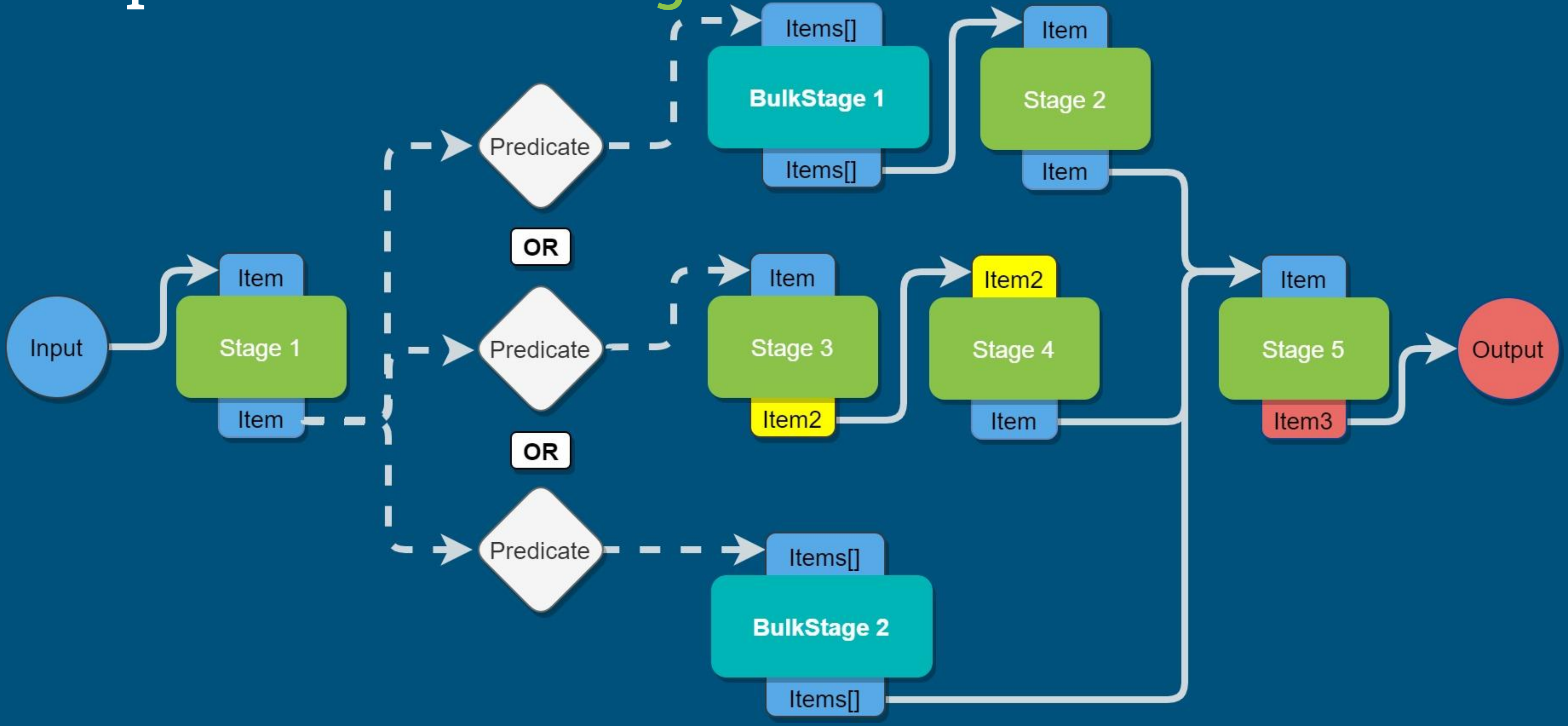
# Pipeline. Single stage options.



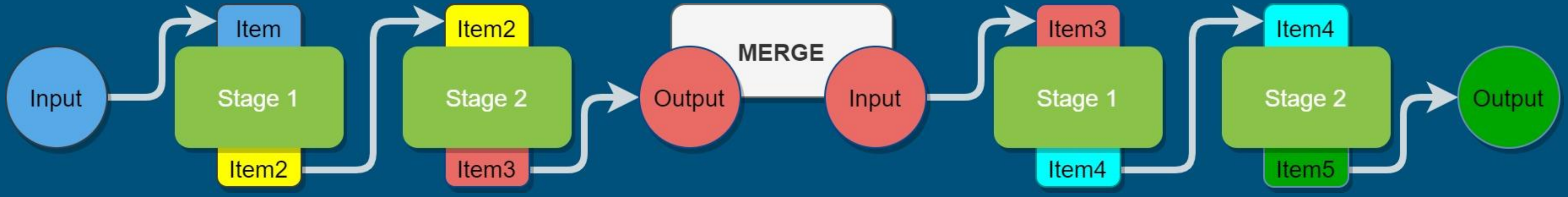
# Pipeline. Exception items.



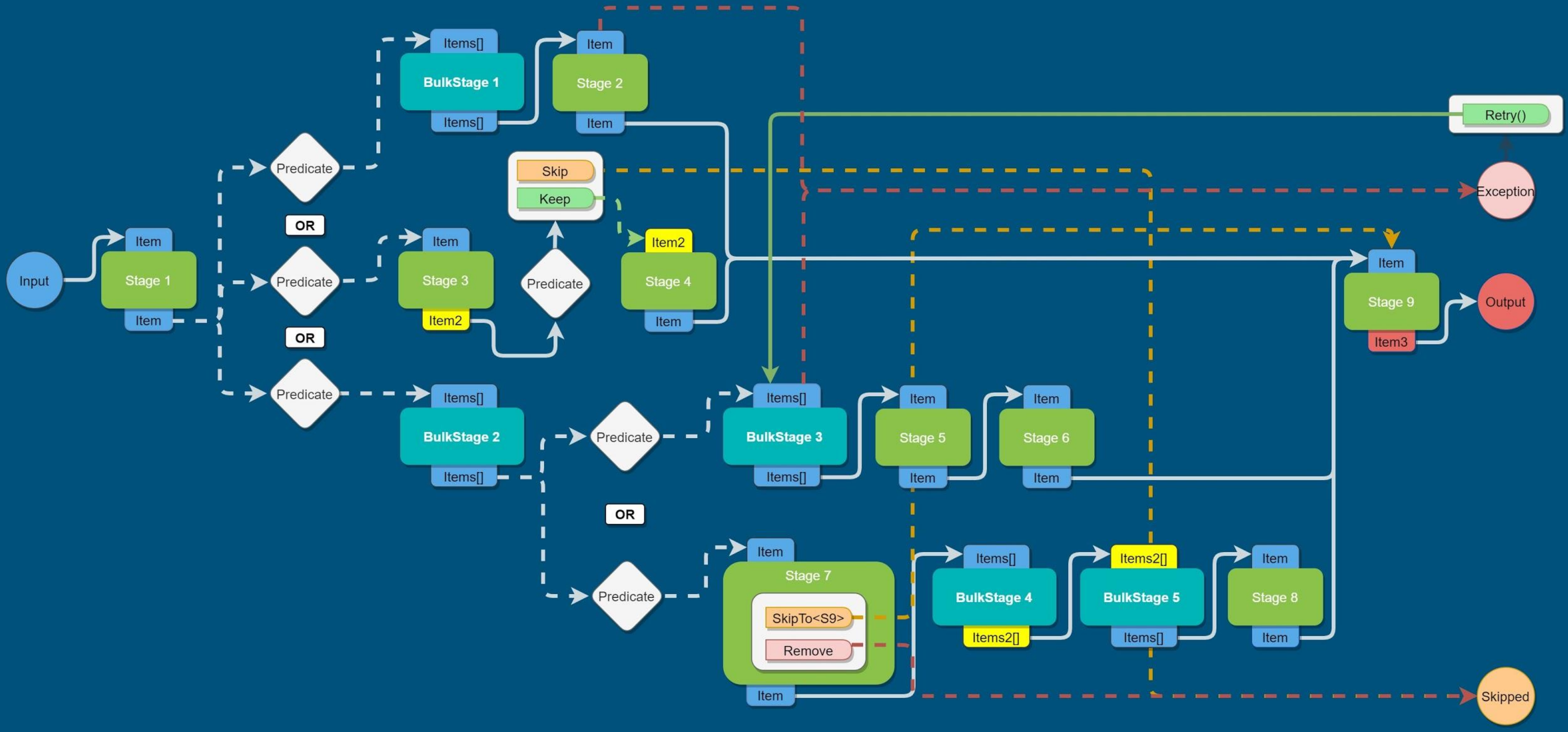
# Pipeline. Branching.



# Pipeline. Merging.



# Pipeline. Complex architecture.

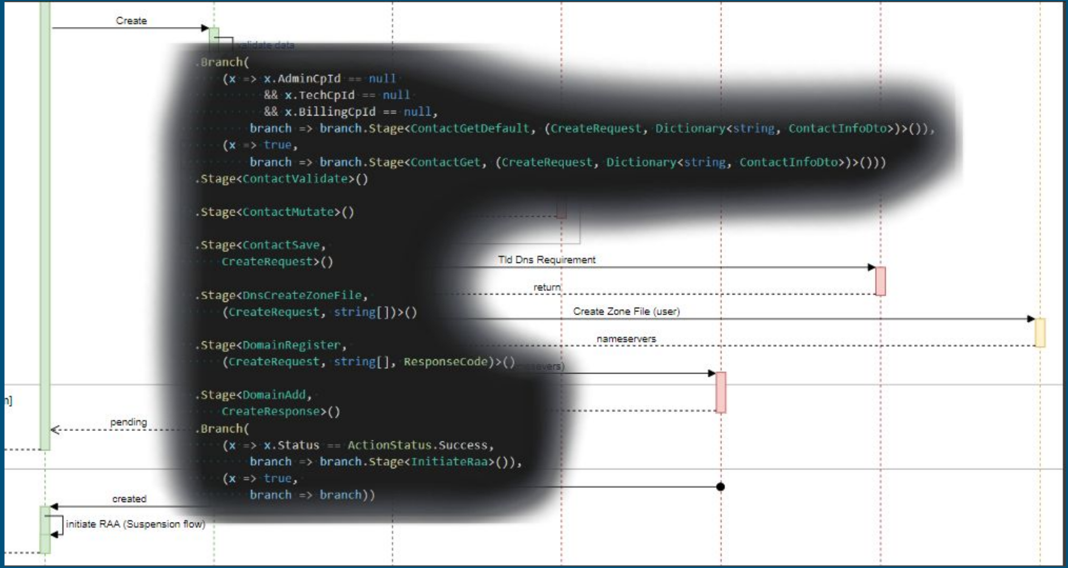


# Pipeline. Complex architecture in code.

```
// Configure stages
var pipelineSetup = PipelineCreator
    .Stage<Stage_1, Item>()
    .Branch(
        (x => indexesForBranch1.Contains(x.Index),
            branch => branch
                .BulkStage<BulkStage_1>()
                .Stage<Stage_2>()),
        (x => indexesForBranch2.Contains(x.Index),
            branch => branch
                .Stage<Stage_3, Item2>()
                .Stage<Stage_4, Item>(x => x.GetItem().Index != 4 ? PredicateResult.Keep : PredicateResult.Skip)),
        (x => true,
            branch => branch
                .BulkStage<BulkStage_2>()
                .Branch(
                    (x => indexesForBranch3SubBranch1.Contains(x.Index),
                        subBranch => subBranch
                            .BulkStage<BulkStage_3>()
                            .Stage<Stage_5>()
                            .Stage<Stage_6>()),
                    (x => true,
                        subBranch => subBranch
                            .Stage<Stage_7>()
                            .BulkStage<BulkStage_4, Item2>()
                            .BulkStage<BulkStage_5, Item>()
                            .Stage<Stage_8>())))
    .Stage<Stage_9, Item3>();
```

# Dataflow. Is it appropriate?

- Development from scratch more natural.
- Parallel software development approach in common, since - Moore's Law.
- GoDaddy uses theirs implementation - “Thespian” for Python github. 😅



```
.Branch(
    (x => x.AdminCpId == null
          && x.TechCpId == null
          && x.BillingCpId == null,
        branch => branch.Stage<ContactGetDefault, (CreateRequest, Dictionary<string, ContactInfoDto>)>()),
    (x => true,
        branch => branch.Stage<ContactGet, (CreateRequest, Dictionary<string, ContactInfoDto>)>()))
.Stage<ContactValidate>()

.Stage<ContactMutate>()

.Stage<ContactSave,
    CreateRequest>()

.Stage<DnsCreateZoneFile,
    (CreateRequest, string[])>()

.Stage<DomainRegister,
    (CreateRequest, string[], ResponseCode)>()

.Stage<DomainAdd,
    CreateResponse>()
.Branch(
    (x => x.Status == ActionStatus.Success,
        branch => branch.Stage<InitiateRaa>()),
    (x => true,
        branch => branch))
```

## More than words

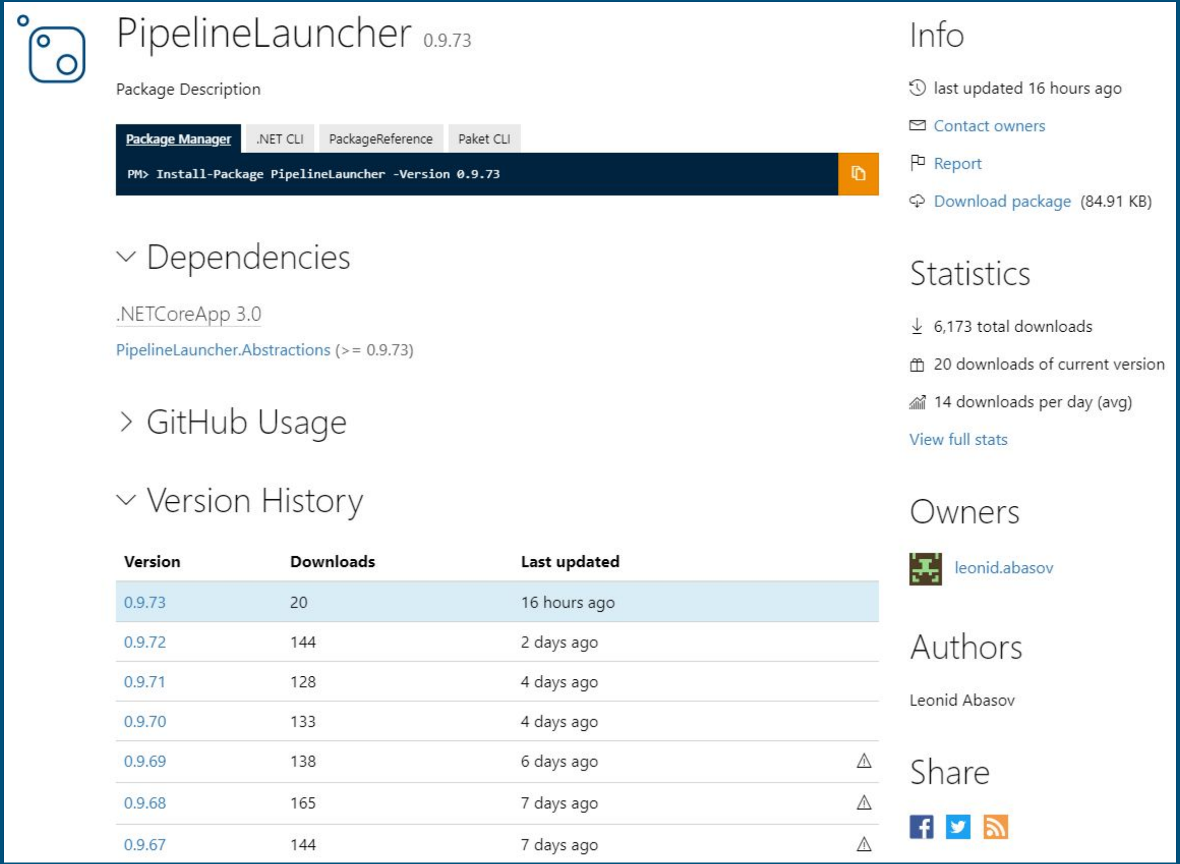
Let it be shown.

- PipelineLauncher DEMO.
- Performance tests.
- Demo usages:
  - Features
  - Configurations.
  - Pipelines
  - Extensions

## Resources. Or rather, a list of them.

- Dataflow (Task Parallel Library) on MSDN
- Dataflow architecture Wikipedia.
- TPL DF by Example: Dataflow and Reactive Programming in .Net. on Amazon.
- Dataflow and Reactive Programming Systems: A Practical Guide on Amazon.
- TPL Dataflow Tour on Channel 9.
- Actor Model in Wikipedia.
- Использование TPL Dataflow для многопоточной компрессии файлов habr.

# Incredible support from developers:



PipelineLauncher 0.9.73

Package Description

Package Manager | .NET CLI | PackageReference | Paket CLI

```
PM> Install-Package PipelineLauncher -Version 0.9.73
```

## Dependencies

.NETCoreApp 3.0

PipelineLauncher.Abstractions (>= 0.9.73)

## GitHub Usage

## Version History

| Version | Downloads | Last updated | |
|---|---|---|---|
| 0.9.73 | 20 | 16 hours ago | |
| 0.9.72 | 144 | 2 days ago | |
| 0.9.71 | 128 | 4 days ago | |
| 0.9.70 | 133 | 4 days ago | |
| 0.9.69 | 138 | 6 days ago | ⚠ |
| 0.9.68 | 165 | 7 days ago | ⚠ |
| 0.9.67 | 144 | 7 days ago | ⚠ |

## Info

last updated 16 hours ago

Contact owners

Report

Download package (84.91 KB)

## Statistics

6,173 total downloads

20 downloads of current version

14 downloads per day (avg)

View full stats

## Owners

leonid.abasov

## Authors

Leonid Abasov

## Share

## Links. Sources.

- PipelineLauncher nuget
- PipelineLauncher.Demo github.

# Thanks

for attention.

# Reactive Extensions (Rx). Basics.

**The Reactive Extensions (Rx)** is a library for composing asynchronous and event-based programs using observable sequences and LINQ-style query operators. Using Rx, developers represent asynchronous data streams with Observables, query asynchronous data streams using LINQ operators, and parameterize the concurrency in the asynchronous data streams using Schedulers.

Simply put , Rx = Observables + LINQ + Schedulers.