Access, Automation, Analytics, AI

Arash Abadpour - arash@abadpour.com

April 11, 2025

"... the four A's that we're after ... (1) Accessibility when I ask a question I want to be able to access the data that allows me to answer the question that I'm asking (2) Automation our ability to make routine tasks that are presently done by humans so that they can be done by machines ... (3) Analytics we want to be able to generate insights that might not otherwise be obvious to us (4) AI ... - *James C. Slife*, The Future of Warfare: Preparing U.S. Military Forces for Competition and Contestation, GSF 2024 [1]."

Abstract

First, we develop a mathematical model to discuss the "Four A's" in Section 1. Then, we propose expansions for Access and Automation in Sections 2 and 3, respectively. We briefly review a proposed view of Analytics as Access to the outputs of Automation in Section 4. Finally, in Section 5, we review a reference implementation of the proposed framework [2] based on *Bash* [3] expansions that call into *Python* [4] in an AI application.

Contents

1 Theoretical Framework

Hypergraph of Objects & Commands

A group of *operators* maintain a growing space of *commands* in a set of *repositories*, using a system such as *git* [5] and following a collective peer-reviewed *pull-request* [6] process. Each operator can access a set of *machines* on a system such as *SageMaker* [7] and create *shells* on them to run commands to produce *objects* which have an implicit or explicit value.

The objective of every operator is to increase the quality and quantity of the objects they generate. Therefore, the operators are interested in mechanisms that enable them to build the commands in ways that allow them to generate large quantities of objects that adhere to requirements that are known in the future from objects that may not exist yet. This is particularly important because commands run other commands and some objects are intermediaries in generating other objects.

As such, we model an AI system as an infinite *hypergraph* [8] where the nodes are the objects and the edges are the commands. We note that this is a Hypergraph, and not a regular graph, because any edge can modify or create zero or more objects from zero or more existing objects. One subset of commands take in zero objects and generate zero objects. Instead, these commands modify the *state* of one or more machines or shells, and thus participate in the generation of the objects further down the line.

Whereas, a conventional hypergraph generalizes by allowing the edges to connect any two subsets of nodes, here, we are also interested in a second generalization that allows an edge to "call" other edges. An important example of this process is when an operator builds an algo (an edge) and then

submits a command to *AWS Batch* [9] that calls the algo 10,000 times on separate subsets of objects. Similarly, an operator may call a command that deploys the algo as an API that is called on a stream of incoming objects.

Hypergraphs have been used in the past to model parallel data structures [10], for data mining [11], and clustering [12]. For a list of other relevant uses of hypergraphs refer to [8]. Moreover, the focus of this work is the development of mathematical tools for the optimal combination of algorithms as black boxes towards practical objectives. This is different from optimizing the internal workings of the same algorithms [13, 14].

State Machines

A machine is a state machine that is connected to many other machines and shares some of its state with them for read and write. A shell is a stateful access mechanism to a machine that an operator uses to run commands. In a 2022 survey of developers, 89% responded that they have a terminal open at least half of the day [15]. Running a command in a shell can potentially modify the state of all other machines. Two examples of machines are a Raspberry Pi [16] that runs Linux and is connected to the AWS infrastructure [17] and a docker container [18] running in AWS Batch. GNU Bash [3] is an example of a shell.

The operators act asynchronously while communicating with each other. Multiple operators may simultaneously use the same machine, and the same operator may simultaneously use multiple machines. Only one operator uses a shell at one time. Some machines are exogenous to this model, yet the operators can access their states in read or write modes through running commands. Cloud storage [19] and compute resources [9] are examples of these machines.

We, therefore, recognize the existence of a plurality of interconnected state machines. Objects are artifacts on some of these machines, and, therefore, their content is a component of the state of the machine(s) that carry them. Hence, the hypergraph is a subset of the state of the universal state machine.

Commands & Expansions

A command is a string of characters that is meaningful to *Bash* [3]. Bash is a "Unix shell and command language first released in 1989 that has been used as the default login shell for most Linux distributions" [20]. A shell is a "macro processor that executes commands" [21], where "macro processor means functionality where text and symbols are expanded to create larger expressions" [21]. There are seven kinds of *expansions* [22] in Bash.

Expansions

Brace Expansion [23] expands '`a{d,c,b}e`' to '`ade ace abe`'. *Tilde Expansion* [24] relates to words that begin with an unquoted tilde character (~). *Parameter and Variable Expansion* [25] enable the use of variables, as `${variable}`, as well as more elaborate pattern matching forms such as `${parameter/#pattern/string}`. *Command Substitution* "allows the output of a command to replace the command itself" [26]. *Arithmetic expansion* [27] enables arithmetic operations using the form `$(( expression ))` and *Word Splitting* [28] governs the splitting of the command to words. Finally, *Filename Expansion* [29] enables the familiar wildcard reference to filenames using '*' and '?'.

Command Syntax

We are interested in a special category of valid bash commands [30] that start with a specially–crafted callable, continue with a prescribed sequence of identifiers, and end with arguments. A *callable* is a valid command with no space and control operators [31]. Some of the well-known callables are *git* [5], *docker* [18], *pushd* [32], and *nano* [33].

Callables

In this convention, the type of each identifier is known based on the command until that identifier, and is one of the following.

- A `value`, either numerical or a filename, for example.
- An *options*.
- An object name or pointer.

This is an example command,

```
vanwatch ingest \
    area=vancouver,~batch,count=5,gif . \
    --count 12
```

Here, "`vanwatch`" is the callable and "`ingest`" is the task, effectively an Enum, which is an `Options`. The command continues with "`area=vancouver, batch,count=5,gif`", which is an `Options`, ".", which is an object pointer, and "`--count 12`", which are the arguments.

Conventions

argparse [34], *click* [35], *fire* [36], and many other command line parsers support the `--<keyword> <value>` convention. These arguments are captured in commands by ending calls to Python with

`--<keyword> <value>`

"$@:<number>". Command substitution [26] is useful where a delimited list of keywords is generated and consumed, such as in `for` loops. We recommend controlling these lists using the three arguments `--count`, `--delim`, and `--offset`.

`--delim space`

```
local object_name
for object_name in $(<command-1> \
    --count 2 \
    --offset 1 \
    --delim space); do
    <command-2> $object_name <args>
    [[ $? -ne 0 ]] && return 1
done
```

`@env`

System state is carried in a set of environment variables that are generaly `export`ed during initialization and are used to harmonize paths and other machine–specific parameters. These variables are listed by `@env [<keyword>]`. `@help <command>` shows help about `<command>`. `@init <options>` initialize the core, and therefore all plugins, and `<plugin-name> init <options>` initializes `<plugin-name>`.

`@help` `@init`

2 Access

Callable Expansion

When the callable `<func>` receives a command that starts with the task `<task>`, it calls the function `<func>_<task>` with the rest of the command, if such function exists. For example, here, the callable `bluer_ai_conda`, which is aliased to `@conda`, is defined.

```
#! /usr/bin/env bash

function bluer_ai_conda() {
    local task=$1

    local function_name=bluer_ai_conda_$task
    if [[ $(type -t $function_name) == "function" ]]; then
        $function_name "${@:2}"
        return
    fi

    conda "$@"
}

bluer_ai_source_caller_suffix_path /conda
```

This mechanism works with the function `bluer_ai_conda_exists`, which resides in `/conda/exists.sh`, listed below,

```
#! /usr/bin/env bash

function bluer_ai_conda_exists() {
    local options=$1
    local environment_name=$(bluer_ai_option "$options" name bluer_ai)

    if conda info --envs | grep -q "^$environment_name "; then
        echo 1
    else
        echo 0
    fi
}
```

The result is the *super-command* `@conda` which behaves as `conda`, except when the function `bluer_ai_conda_<task>` is defined. Here is an example of the outcome assembled together,

```
[[ $(@conda exists name=<env-name>) == 1 ]] && echo "found."
```

We use this *namespacing* [37] mechanism for organization as well as orchestration. In practice, this expansion yields callables with multiple prefixes. We generate `@<keyword>` aliases [38] to facilitate user access, as stated above. In practice, the use of this expansion is comparable to fitting the spanning tree of an Application Programming Interface (API) [39] to a dataset of use of digital tools by operators recorded as sequences of keystrokes and mouse clicks.

Options Expansion

An options is a string representation of a dictionary, such as,

```
<keyword-1>=<value-1>,<keyword-2>=<value-2>,...,<keyword-3>,-<keyword-4>},...
```

Options is implemented using basic Python and, therefore, the *options expansion* is available to Bash commands through command substitution [26]. In practice, the two additional expansions `@option::int` and `@option::choice` cast the output to an integer and select it from a list (equivalent to an Enum [40]), respectively.

```
value=$(@option "$options" <keyword> [<default>])

value=$(@option::int "$options" <keyword> 0 | 1)

value=$(@option::choice "$options" <comma,separated,list> <default>)
```

Object Expansion

Commands manipulate data as objects. An object is a uniquely named collection of files and folders and can be downloaded or uploaded, in part or as a whole. Some objects already exist in the environment. For example, an *item* in a *STAC collection* [41] (a datacube) or a dataset in *Kaggle* [42] are objects. A curated dataset, a model trained on it, and the model's predictions on a datacube, are examples of other objects.

Object Pointers

An object may be *selected*,

```
@select <object-name>
```

When `<object-name>` is selected, '`.`' expands to `<object-name>`. Similarly, '`..`', '`...`', and so on, as deep as needed, expand to the names of the previously selected object and the one before that. Commands default the objects they consume and modify to '`.`', '`..`', and so on. Therefore, because the commands in a script generally use the same objects, selecting the objects enables their names to be replaced with pointers. Often the defaults of the commands are designed to enable the omission of the pointers as well.

Object Metadata

Every object carries a `metadata.yaml` file that can be `get` and `post` as a dictionary.

```
@metadata get key=<key> [.|<object-name>]

@metadata post <key> <value> - [.|<object-name>]
```

Object Tags

An object can have many tags. A tag is key-value pair that is maintained in a system such as *MLflow* [43] and is `set` and `get`, and can be `search`ed,

```
@mlflow tags get [.|<object-name>] [--tag <tag>]

@mlflow tags search [<keyword-1>=<value-1>,<keyword-2>,~<keyword-3>]

@mlflow tags set [.|<object-name>] [<keyword-1>=<value>,<keyword-2>,~<keyword-3>]
```

Object Persistence

Object are persisted on AWS S3 [19]. Objects can be downloaded, uploaded, and listed,

```
@download [filename=<filename>] [.|<object-name>]

@upload [filename=<filename>] [.|<object-name>]

@list cloud|local [.|<object-name>]
```

3 Automation

Prefixing

A `<prefix>` can run a `<command>` with certain `<options>` through the *prefixing expansion*,

```
<prefix> <options> <command>
```

This expansion enables submitting a command to another machine, including a *docker* [18] container, and compute resources, such as *AWS Batch* [9] and Argo Workflows [44].

```
@eval [dryrun,path=<path>] <command>

@docker eval [dryrun] <command>

@batch eval [dryrun,name=<job-name>] <command>
```

`workflows`

A `workflow` is a set of commands that depend on each other because they consume objects that are generated by other commands, among other reasons. We model a workflow as a Directed Acyclic Graph (DAG), where each node is a command. We use this abstraction through `NetworkX` [45] to run workflows on AWS Batch [9] and Argo Workflows [44]. In practice, there is often a series of commands that generate a cascade of objects that contains the final products. Each command in this series has one or more options and takes in and produces zero or more objects. Here is a representative case,

Cascading Options

```
vanwatch \
    ingest \
    target=<target>,count=<count> \
    $object_name

vanwatch \
    detect \
    count=<count>,gif,model=<model-id>,publish \
    $object_name \
    --overwrite 1
```

The *options cascade expansion* compresses this cascade of commands as follows,

```
vanwatch \
    ingest \
    target=<target>,count=<count> \
    $object_name \
    detect,count=<count>,gif,model=<model-id>,publish \
    --overwrite 1
```

List of Objects

A series of objects that are tagged by a unique tag to represent a list of objects. Or, the list of objects may be maintained in the metadata of an object. This object or tag expands to the list of objects, through tag or metadata mechanisms.

Terraforming

Terraforming is the process of installing modules and adjusting the configuration of a shell or a machine for a specific purpose through running a series of commands. These commands are often long and unintuitive and contain secrets and other environment variables. Terraforming is generally persistent and has to be done once at the first use of a machine. Examples include `ssh` into a new machine, starting a machine on a service such as AWS SageMaker [7], and running commands inside the Python Console in *QGIS* [46].

`@seed`

The *seed expansion* generates the code required for terraforming a target and transfers it through one of the listed mechanisms,

```
@seed [<target>] [clipboard|filename=<filename>|usb-key|scp|screen]
```

4 Analytics

We understand Analytics as Access to the outcomes of Automation. See Sections 2 and 3.

5 AI

`vanwatch` [47] discovers traffic cameras in a target, ingests images from them, and runs detection algo on the images to generate time series analytics.

`discover`

Cameras are discovered through public websites and, therefore, they are represented in different formats in different targets. `vanwatch discover` discovers the cameras in `<target>` and stores them in `<target>.geojson` in the object `<object-name>` and tags the object for discovery by `ingest`.

```
vanwatch \
    discover \
    [target=<target>,count=<-1>,dryrun,~tag,~upload] \
    [-|<object-name>]
```

Here, `target=<target>,count=<-1>,dryrun, tag, upload` is an options. If the object pointer `-` is selected, a timestamped object prefixed `<target>-discover` is generated. This options input also enables `dryrun`, a limited count, and the ability to disable tagging and uploading the object. These features are handy for testing and special runs.

`vanwatch discover` points to the function `vancouver_watching_discover`[1] through callable expansion and the alias `vanwatch`.

```
#! /usr/bin/env bash

function vancouver_watching_discover() {
    local options=$1
    local target=$(bluer_ai_option "$options" target vancouver)
    local do_dryrun=$(bluer_ai_option_int "$options" dryrun 0)
    local do_tag=$(bluer_ai_option_int "$options" tag $(bluer_ai_not $do_dryrun))
    local do_upload=$(bluer_ai_option_int "$options" upload $(bluer_ai_not $do_dryrun))
```

`options` uses options expansion.

```
    local object_name=$(bluer_ai_clarify_object $2 \
        $target-discover-$(bluer_ai_string_timestamp_short))
```

`object_name` uses object expansion.

```
    local function_name=vancouver_watching_discover_$target
    if [[ $(type -t $function_name) != "function" ]]; then
        bluer_ai_log_error "vancouver_watching: discover: $target: target not found."
        return 1
    fi
```

A callable expansion.

```
    bluer_ai_clone \
        ~relate,~tags \
        $VANWATCH_QGIS_TEMPLATE \
        $object_name

    bluer_ai_log "discovering $target -> $object_name"
    bluer_ai_eval ,$options \
        $function_name \
        ,$options \
        $ABCLI_OBJECT_ROOT/$object_name \
        "${@:3}"
    local status="$?"

    [[ "$do_upload" == 1 ]] &&
        bluer_objects_upload - $object_name

    [[ "$status" -ne 0 ]] && return $status

    [[ "$do_tag" == 1 ]] &&
        bluer_objects_mlflow_tags set \
```

[1] https://github.com/kamangir/vancouver-watching/blob/main/vancouver_watching/.abcli/discover.sh

```
            $object_name \
            app=vancouver_watching,target=$target,stage=discovery

    return $status
}

bluer_ai_source_caller_suffix_path /discovery
```

The callable expansion above points to the function `vancouver_watching_discover_vancouver`[2].

```
#! /usr/bin/env bash

function vancouver_watching_discover_vancouver() {
    local options=$1
    local object_path=$2

    curl \
        https://opendata.vancouver.ca/explore/dataset/web-cam-url-links/download/?format=geojson \
        >$object_path/detections.geojson

    local count=$(bluer_ai_option_int "$options" count -1)

    bluer_ai_eval ,$options \
        python3 -m vancouver_watching.discover \
        discover_cameras_vancouver_style \
        --filename $object_path/detections.geojson \
        --prefix https://trafficcams.vancouver.ca/ \
        --count $count \
        "${@:3}"
}
```

`vanwatch ingest` finds the latest set of cameras discovered in `<target>` through tag search and ingests `count` images into `<object-name>` and then runs `vanwatch detect` unless `detect` is present. `object-name` is then tagged for future discovery.

ingest

```
vanwatch \
    ingest \
    [target=<target>,count=<-1>,~download,dryrun,~upload] \
    [-|<object-name>] \
    [detect,count=<-1>,~download,dryrun,gif,model=<model-id>,publish,~upload] \
    [--overwrite 1] \
    [--verbose 1]
```

`vanwatch ingest` generates an object from another that it finds through tag search. Note that this command uses arguments as well as two cascading options. The first controls the ingest of the images and the second is passed to `detect`.

detect

`vanwatch detect` runs object detection algo [48] on the images ingested into `<object-name>` and updates `<target>.geojson`.

```
vanwatch \
    detect \
    [count=<-1>,~download,dryrun,gif,model=<model-id>,publish,~upload] \
    [.|<object-name>] \
    [--overwrite 1] \
    [--verbose 1]
```

[2] https://github.com/kamangir/vancouver-watching/blob/main/vancouver_watching/.abcli/discovery/vancouver.sh

References

[1] James C. Slife. The Future of Warfare - Preparing U.S. Military Forces for Competition and Contestation - GSF 2024 - YouTube. Center for Strategic & International Studies, YouTube, https://www.youtube.com/watch?v=cLmcqy5vJv4&t=2161s.

[2] Arash Abadpour. bluer-ai: a language to speak AI. GitHub repository, https://github.com/kamangir/bluer-ai.

[3] Bash - GNU Project - Free Software Foundation. https://www.gnu.org/software/bash/.

[4] Welcome to Python.org. https://www.python.org/.

[5] Git - git Documentation. https://git-scm.com/docs/git.

[6] About pull requests - GitHub Docs. https://docs.github.com/en/pull-requests/collaborating-with-pull-re

[7] Machine Learning Service - Amazon SageMaker - AWS. https://aws.amazon.com/sagemaker/.

[8] Alain Bretto. *Hypergraph Theory; An Introduction*. Springer, 2013.

[9] Efficient Batch Computing - AWS Batch - AWS. https://aws.amazon.com/batch/.

[10] B. Hendrickson and T.G. Kolda. Graph partitioning models for parallel computing. *Parallel Computation*, 26:1519–1545, 2000.

[11] C. Hébert, A. Bretto, and B. Crémilleux. A data mining formalization to improve hypergraph minimal transversal computation. *Fundamenta Informaticae*, 80(4):415–433, 2007.

[12] S.R. Bulò and M. Pelillo. A game-theoretic approach to hypergraph clustering. *proceedings of the NIPS*, pages 1571–1579, 2009.

[13] Gitta Kutyniok. The mathematics of artificial intelligence, 2022.

[14] Gabriel Peyré. The mathematics of artificial intelligence, 2025.

[15] Will McGugan. Why I Founded Textualize. https://www.textualize.io/blog/why-i-founded-textualize/, 2023. Accessed: 2025-01-01.

[16] Teach, learn, and make with the Raspberry Pi Foundation. https://www.raspberrypi.org/.

[17] Cloud Computing Services - Amazon Web Services (AWS). awshttps://aws.amazon.com/.

[18] Use the Docker command line - Docker Docs. https://docs.docker.com/engine/reference/commandline/cli/.

[19] Amazon S3 - Cloud Object Storage - AWS. https://aws.amazon.com/s3/.

[20] Bash (unix shell) - wikipedia. https://en.wikipedia.org/wiki/Bash_(Unix_shell).

[21] What is a shell? (Bash Reference Manual). https://www.gnu.org/software/bash/manual/html_node/What-is-a-

[22] Shell Expansions (Bash Reference Manual). https://www.gnu.org/software/bash/manual/html_node/Shell-Expa

[23] Brace Expansion (Bash Reference Manual). https://www.gnu.org/software/bash/manual/html_node/Brace-Expa

[24] Tilde Expansion (Bash Reference Manual). https://www.gnu.org/software/bash/manual/html_node/Tilde-Expa

[25] Shell Parameter Expansion (Bash Reference Manual). https://www.gnu.org/software/bash/manual/html_node/Sh

[26] Command Substitution (Bash Reference Manual). https://www.gnu.org/software/bash/manual/html_node/Comma

[27] Arithmetic Expansion (Bash Reference Manual). https://www.gnu.org/software/bash/manual/html_node/Arithm

[28] Word Splitting (Bash Reference Manual). https://www.gnu.org/software/bash/manual/html_node/Word-Splitti

[29] Bash Startup Files (Bash Reference Manual). https://www.gnu.org/software/bash/manual/html_node/Filename-

[30] Shell Syntax (Bash Reference Manual). https://www.gnu.org/software/bash/manual/html_node/Shell-Syntax.

[31] Definitions (Bash Reference Manual): control operator. https://www.gnu.org/software/bash/manual/html_node/

[32] Directory Stack Builtins (Bash Reference Manual). https://www.gnu.org/software/bash/manual/html_node/Dire

[33] nano - Text editor. https://www.nano-editor.org/.

[34] argparse — Parser for command-line options, arguments and sub-commands — Python 3.12.4 documentation. https://docs.python.org/3/library/argparse.html.

[35] Click - The Pallets Projects. https://palletsprojects.com/p/click/.

[36] Python Fire. https://google.github.io/python-fire/.

[37] Namespace - Wikipedia. https://en.wikipedia.org/wiki/Namespace#Emulating_namespaces.

[38] Aliases (Bash Reference Manual). https://www.gnu.org/software/bash/manual/html_node/Aliases.html.

[39] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. University of California, Irvine, 2000.

[40] enum — Support for enumerations — Python 3.12.4 documentation. https://docs.python.org/3/library/enum.html.

[41] SpatioTemporal Asset Catalog (STAC) Specification. Introduction to STAC. Accessed: 2024-12-22.

[42] Qi Chen, Lei Wang, Yifan Wu, Guangming Wu, Zhiling Guo, and Steven L Waslander. Aerial imagery for roof segmentation: A large-scale dataset towards automatic mapping of buildings. *ISPRS Journal of Photogrammetry and Remote Sensing*, 147:42–55, 2019.

[43] MLflow. https://mlflow.org/.

[44] Argo Workflows - Kubernetes-native workflow engine supporting DAG and step-based workflows. https://argoproj.github.io/workflows/.

[45] NetworkX Developers. NetworkX. https://networkx.org/, 2023.

[46] 4. Features — QGIS Documentation documentation. https://docs.qgis.org/3.34/en/docs/user_manual/preambl

[47] Arash Abadpour. Vancouver Watching: Vancouver watching with AI. GitHub repository, https://github.com/kamangir/Vancouver-Watching.

[48] Ultralytics HUB - YOLOv8x. https://hub.ultralytics.com/models/R6nMlK6kQjSsQ76MPqQM?tab=preview.

built by `gizai-7.332.1`.