1 Overview

primUR is a command-line pipeline for identifying lineage-specific genomic regions and deriving diagnostic PCR primers from unique genomic regions (URs) for a set of target taxa. The pipeline operates on whole-genome data and follows a fixed sequence of stages:

1. parse and validate user input,
2. acquire target and neighbor genomes,
3. reconstruct phylogeny,
4. identify lineage-specific regions,
5. design and evaluate diagnostic primers.

All stages are implemented in a single shell script that can be executed directly or reconstructed from this literate source. The following top-level structure outlines the execution flow of the pipeline.

1.1 Command-line interface and initialization

The first part of the script defines the public command-line interface, initializes default settings, parses user-provided arguments, validates inputs, and verifies the availability of all external dependencies. Logging and runtime tracking are also initialized at this stage.

⟨*primUR.sh*⟩≡
⟨*cli: shebang*⟩
⟨*cli: usage function*⟩

⟨*setup: defaults*⟩
⟨*setup: log and dependency helpers*⟩
⟨*cli: parse arguments*⟩
⟨*setup: validate options and inputs*⟩
⟨*setup: check dependencies*⟩
⟨*setup: initialize logs*⟩

1.2 Data acquisition and analysis loop

After initialization, the pipeline iterates over all target taxa listed in the query file. For each target, a dedicated working directory is created, genome data are acquired if necessary, and all downstream analyses are performed in sequence. Control-flow options allow execution to stop after phylogenetic reconstruction or to resume from that point.

⟨*primUR.sh*⟩+≡
⟨*data: acquisition main loop*⟩

1.3 Finalization

Once all targets have been processed, the pipeline records an end timestamp and reports completion. Runtime information is appended to the timer log, providing a minimal execution record for reproducibility.

⟨*primUR.sh*⟩+≡
⟨*pipeline: end-of-run timer log*⟩

2 Command-line interface and setup

2.1 Synopsis

`primUR` is executed as a shell script. The required inputs are a query file listing target species names (one per line), a genome neighborhood database used by `taxi` and `neighbors`, and a database used by `scop`. Additional options control verbosity and whether the pipeline is stopped after tree plotting or resumed from that point.

2.2 Usage

The following usage text is printed by `primUR -h` and defines the public interface of the pipeline.

⟨*cli: shebang*⟩≡

```
#!/bin/bash
```

⟨*cli: usage function*⟩≡
```
usage() {
cat <<'USAGE'
Usage:
primUR -t <query.txt> -n <neidb_path> -d <scop_db_path> [options]

Required:
-t FILE   Query file (one target species per line)
-n PATH   Genome database path for ‘taxi‘ and ‘neighbors‘
-d PATH   NCBI nt/SCOP database path for ‘scop‘
Options:
-f        Feedback to terminal (quiet by default)
-c        Run up to (and including) tree plotting, then stop
-r        Resume after tree plotting (expects prior outputs)
-h        Show this help

Examples:
primUR -t query.txt -n ~/DBs/neidb -d ~/DBs/nt
primUR -f -c -t query.txt -n /data/neidb -d /data/nt
USAGE
}
```

2.3 Defaults and helper functions

By default, `primUR` runs quietly and only prints status output when feedback mode (`-f`) is enabled. The helper function `need` terminates execution if a required external command is missing.

⟨*setup: defaults*⟩≡
```
QUIET=true
CHECK=false
RESUME=false
QUERY=""
DB=""
SCOPDB=""
```

⟨*setup: log and dependency helpers*⟩≡
```
log() { $QUIET && return 0; printf '%s\n' "$*"; }
need() { command -v "$1" >/dev/null 2>&1 || \
{ echo "Missing dependency: $1" >&2; exit 1; }; }
```

2.4 Argument parsing

Arguments are parsed using a `case` statement. Required arguments populate the variables `$QUERY`, `$DB`, and `$SCOPDB`. Options `-c` and `-r` are mutually exclusive.

⟨*cli: parse arguments*⟩≡
```
while [[ $# -gt 0 ]]; do
  case "$1" in
    -t) QUERY="${2-}"; shift 2 ;;
    -n) DB="${2-}";    shift 2 ;;
    -d) SCOPDB="${2-}"; shift 2 ;;
    -f) QUIET=false;   shift ;;
    -c) CHECK=true;    shift ;;
    -r) RESUME=true;   shift ;;
    -h) usage; exit 0 ;;
    --) shift; break ;;
    -*) echo "Unknown option: $1" >&2; usage; exit 1 ;;
    *)  echo "Unexpected positional argument: $1" >&2; usage; \
        exit 1 ;;
  esac
done
```

2.5 Input validation

The pipeline terminates early if mutually exclusive options are chosen, if required inputs are missing, or if the query file is not readable.

⟨*setup: validate options and inputs*⟩≡
```
if $CHECK && $RESUME; then
  echo "Choose either -c or -r, not both." >&2
  exit 1
fi

[[ -z "$QUERY"  ]]  && { echo "Error: -t <query.txt> \
   is required." >&2; usage; exit 1; }
[[ -z "$DB"     ]]  && { echo "Error: -n <neidb_path> \
   is required." >&2; usage; exit 1; }
[[ -z "$SCOPDB" ]]  && { echo "Error: -d <scop_db_path> \
   is required." >&2; usage; exit 1; }

[[ -r "$QUERY" ]] || { echo "Error: query file not readable: \
   $QUERY" >&2; exit 1; }
```

2.6 Dependencies

Before starting the analysis, `primUR` verifies that all external commands required for the pipeline are available in `$PATH`. This makes failures deterministic and moves missing-dependency errors to the start of execution.

⟨*setup: check dependencies*⟩≡

```
for cmd in taxi neighbors datasets unzip phylonium nj midRoot \
                land plotTree sed awk tr sort head tail \
                makeFurDb fur cleanSeq cres fa2prim \
                primer3_core prim2tab scop; do
  need "$cmd"
done
```

2.7 Logging and timer initialization

The pipeline writes tool output and runtime metadata to the `logs/` directory. This includes separate logs for downloads, phylogenetic distance computation, and marker discovery, as well as a timer log recording the start time.

⟨*setup: initialize logs*⟩≡

```
mkdir -p logs
> logs/datasets.log
> logs/phylonium.log
> logs/fur.log
{
  echo "Start:"
  date
} > logs/timer.log
log "$(cat logs/timer.log)"
```

3 Data acquisition

This stage prepares a working directory for each queried target taxon and downloads all genomes required for downstream analysis. For every target, the pipeline performs four tasks:

1. create a target-specific output directory structure,
2. resolve the target taxon name to a taxonomy identifier,
3. derive two accession lists (targets and neighbors) using the neighborhood definition,
4. download and extract the corresponding genome FASTA files into standardized locations.

If resume mode is enabled, this stage can be skipped to avoid re-downloading already retrieved genomes.

⟨*data: acquisition main loop*⟩≡

```
while read -r target; do
  ⟨data: per-target header⟩
  ⟨data: per-target directory setup⟩
  ⟨data: resolve target and download genomes unless RESUME⟩

  if ! $RESUME; then
      ⟨tree: compute distance matrix⟩
      ⟨tree: infer and root tree⟩
      ⟨tree: normalize labels⟩
      ⟨tree: plot⟩

      $CHECK && { log "--check set; stopping after tree plotting for $target";
          continue; }
  fi

  ⟨markers: build fur database⟩
  ⟨markers: run fur and clean candidates⟩
  ⟨markers: log and run cres⟩

  ⟨primers: announce primer stage⟩
  ⟨primers: generate and collect candidates⟩
  ⟨primers: report count and outputs⟩
  ⟨primers: save best primer pair⟩
  ⟨primers: run scop⟩

  ⟨pipeline: per-target completion log⟩
done < "$QUERY"
```

The following chunk implements the per-target header and derives a file-system-safe identifier used for directory names and output files.

⟨*data: per-target header*⟩≡
```
log "$target:"
safe_target="${target// /_}"
```

Each target receives its own directory. Within it, target genomes and neighbor genomes are stored separately. This separation is preserved throughout the pipeline to make it explicit which genomes define the presence set and which define the exclusion set.

⟨*data: per-target directory setup*⟩≡
```
mkdir -p "$safe_target" "$safe_target/targets" \
      "$safe_target/neighbors"
```

Genome acquisition is performed only if resume mode is disabled. In non-resume mode, the pipeline first resolves the user-provided target name to a taxonomy identifier, then derives the target and neighbor accession lists, and finally downloads genomes for both lists.

3.1 Resolve target name to taxon id

The pipeline uses `taxi` to query a taxonomy database and extract the taxon id corresponding to the target name. The extracted identifier is stored in a per-target file to preserve provenance and to support manual inspection.

⟨*data: resolve target name to taxon id*⟩≡
```
taxon_id=$(taxi "$target" "$DB" | awk 'NR==2{print $1}')
echo "$taxon_id" >> "$safe_target/taxids.txt"
```

3.2 Compute neighborhood

Given the taxon id, the pipeline computes the local taxonomic neighborhood using `neighbors`. The output is retained in a shell variable and parsed into two sets: a target set and a neighbor set. These sets are later used to download genomes.

⟨*data: compute neighborhood from taxon id*⟩≡
```
neigh_out="$(neighbors -g "$DB" <<< "$taxon_id")"
```

3.3 Write accession lists

The neighborhood output contains entries that are classified into target categories and neighbor categories. The pipeline extracts the accessions, normalizes the delimiter format, and writes them into two files:

- `t_{safe_target}.txt` for targets
- `n_{safe_target}.txt` for neighbors

These files form the authoritative download lists for this stage.

⟨*data: write accession lists for targets and neighbors*⟩≡
```
awk '$1=="t"||$1=="tt"{print $NF}' <<< "$neigh_out" |
tr '|' '\n' > "t_${safe_target}.txt"

awk '$1=="n"{print $NF}' <<< "$neigh_out" |
tr '|' '\n' > "n_${safe_target}.txt"

log "   taxi and neighbors complete"
```

3.4 Download genomes

The pipeline downloads genomes for both the target and neighbor accession lists. Each list is handled identically, except that downloaded genomes are written to different directories and prefixed differently to preserve their provenance.

All download and extraction output from `datasets` and `unzip` is appended to `logs/datasets.log` for transparency and debugging.

⟨*data: download genomes for target and neighbor lists*⟩≡
```
for file in "t_${safe_target}.txt" "n_${safe_target}.txt"; do
  ⟨data: choose output dir and filename prefix⟩
  ⟨data: download genomes for one list file⟩
done
```

For target accessions, genomes are written to `{safe_target}/targets` and prefixed with `t_`. For neighbor accessions, genomes are written to `{safe_target}/neighbors` and prefixed with `n_`.

⟨*data: choose output dir and filename prefix*⟩≡
```
case "$file" in
  t_*) output_dir="$safe_target/targets";   prefix="t_";;
  n_*) output_dir="$safe_target/neighbors"; prefix="n_";;
esac
```

The following loop iterates over each accession in the current list file. For each accession, a genome archive is downloaded using `datasets`, unpacked into a temporary directory, and all `.fna` files within the accession-specific dataset folder are moved into the chosen output directory. Temporary files are removed after each accession to keep the workspace small and avoid accidental cross-contamination between accessions.

⟨*data: download genomes for one list file*⟩≡
```
while read -r accession; do
  ⟨data: download one accession archive⟩
  ⟨data: unzip one accession archive⟩
  ⟨data: move extracted fna files to output⟩
  ⟨data: cleanup one accession temporary files⟩
done < "$file"
```

⟨*data: download one accession archive*⟩≡
```
printf "Downloading: $accession" >> logs/datasets.log
datasets download genome accession "$accession" \
  --filename "${accession}.zip" >> logs/datasets.log 2>&1
```

⟨*data: unzip one accession archive*⟩≡
```
unzip -o "${accession}.zip" -d tmp_unzip \
      >> logs/datasets.log 2>&1
```

The NCBI datasets archive may contain one or more `.fna` files for a given accession. All `.fna` files found in the accession directory are moved into the appropriate target or neighbor directory and renamed by prepending the corresponding prefix.

⟨*data: move extracted fna files to output*⟩≡
```
for fna_file in tmp_unzip/ncbi_dataset/data/"$accession"/*.fna; do
  base=$(basename "$fna_file")
  mv "$fna_file" "$output_dir/${prefix}${base}"
done
```

⟨*data: cleanup one accession temporary files*⟩≡
```
rm -rf tmp_unzip
rm "${accession}.zip"
```

3.5 Outputs of data acquisition

After this stage completes for a given target, the following files and directories are available:

- {safe_target}/targets/ containing downloaded target genome FASTA files
- {safe_target}/neighbors/ containing downloaded neighbor genome FASTA files
- {safe_target}/taxids.txt containing the resolved taxon id(s) used
- t_{safe_target}.txt and n_{safe_target}.txt listing accessions
- logs/datasets.log containing download and extraction logs

These outputs are the inputs to subsequent phylogenetic reconstruction and marker discovery.

4 Phylogenetic reconstruction

For each target, the pipeline constructs a distance-based phylogeny over the target and neighbor genomes that were downloaded during data acquisition. This phylogeny serves two purposes:

- It provides a sanity check that the selected targets and neighbors behave as expected.
- It documents the phylogenetic context used for subsequent marker discovery.

The pipeline computes a genome-wide distance matrix using phylonium, infers a neighbor-joining tree with nj, roots it with midRoot, and normalizes labels using sed. The final tree is saved in Newick format and plotted via plotTree.

If the -c (CHECK) option is used, the pipeline stops after plotting the tree for each target and continues with the next target without performing marker discovery or primer design.

⟨*tree: compute distance matrix*⟩≡

```
phylonium "$safe_target"/targets/* "$safe_target"/neighbors/* \
          > "$safe_target/tree.dist" 2>> logs/phylonium.log
```

The distance matrix is converted into a neighbor-joining tree, rooted, and written to disk in Newick format.

⟨*tree: infer and root tree*⟩≡

```
nj "$safe_target/tree.dist" |
      midRoot |
      land > "$safe_target/tree.nwk" 2>> logs/phylonium.log
```

The tree is post-processed to normalize labels. Single quotes are escaped, and sequence identifiers are reduced to canonical accession-like tokens (e.g. `GCA_...` or `GCF_...`) to improve plot readability.

⟨*tree: normalize labels*⟩≡
```
sed "s/'/''/g" "$safe_target/tree.nwk" |
    sed -E "s/(GC[AF]_[0-9]+\.[0-9]+)[^']*/\1/g" \
        > "$safe_target/tree_clean.nwk"
```

Finally, the normalized Newick tree is plotted.

⟨*tree: plot*⟩≡
```
plotTree "$safe_target/tree_clean.nwk" \
      2>> logs/phylonium.log

      log "   Tree construction complete"
```

4.1 Check and resume control flow

Two options affect control flow at this point:

- `-c` (`CHECK`): stop after tree plotting for each target (skip downstream stages).
- `-r` (`RESUME`): skip data acquisition and tree plotting and continue with marker discovery.

These options are mutually exclusive and validated during argument parsing. Resume mode expects that the per-target directories already contain the downloaded genomes and tree outputs from a previous run.

⟨*data: resolve target and download genomes unless RESUME*⟩≡
```
if ! $RESUME; then
  ⟨data: resolve target name to taxon id⟩
  ⟨data: compute neighborhood from taxon id⟩
  ⟨data: write accession lists for targets and neighbors⟩
  ⟨data: download genomes for target and neighbor lists⟩
  log "   All genome downloads complete"
else
  log "--resume set; starting at marker discovery for $target"
fi
```

5 Marker discovery

After phylogeny construction (or immediately in resume mode), the pipeline performs marker discovery using `fur`. This stage constructs a database from the target and neighbor genome sets and identifies candidate target-specific regions. The candidates are cleaned and written to `max.fasta` for use in primer design.

All `fur` and preprocessing output is directed to log files to support debugging.

⟨*markers: build fur database*⟩≡
```
makeFurDb -t "$safe_target/targets" \
        -n "$safe_target/neighbors" \
        -d "$safe_target/max.db" 2>logs/makeFurDb.log
```

The `fur` results are streamed directly into `cleanSeq`. The resulting FASTA file (`max.fasta`) is the main sequence output of marker discovery.

⟨*markers: run fur and clean candidates*⟩≡
```
fur -d "$safe_target/max.db" 2>logs/fur.log |
  cleanSeq > "$safe_target/max.fasta"
```

A short record of each target's processing is appended to `logs/fur.log`. The `cres` step is run on the candidate FASTA and its output is appended to the same log.

⟨*markers: log and run cres*⟩≡
```
echo "$safe_target" >> logs/fur.log
  cres "$safe_target/max.fasta" >> logs/fur.log
  echo >> logs/fur.log

  log "   fur analysis completed"
```

6 Primer design and filtering

This stage converts candidate marker sequences into PCR primer pairs. Candidate sequences (`max.fasta`) are transformed into primer design input via `fa2prim`, then passed through `primer3_core`. The output is converted to tabular form via `prim2tab`, filtered, and sorted by penalty.

Only primer pairs with penalty ≤ 1 are written to `primers.fasta`. The best scoring primer pair (first 4 lines: header + forward + reverse + blank) is saved as `best_primers.fasta`.

Finally, primers are tested in silico with `scop` against the configured database (`$SCOPDB`), constrained by the target taxon id(s) in `taxids.txt`.

⟨*primers: announce primer stage*⟩≡
```
log "   Running primer3"
```

Primer generation, filtering, and writing are implemented as a single pipeline. The number of retained primer pairs is captured into `count` for reporting.

⟨*primers: generate and collect candidates*⟩≡
```
count=$(
  fa2prim "$safe_target/max.fasta" |
      primer3_core |
      prim2tab |
      tail -n +2 |
      sort -n |
      awk -v out="$safe_target/primers.fasta" '
    $1<=1 {
      printf(">f penalty: %s\n%s\n>r\n%s\n\n", $1,$2,$3) >> out
      n++
    }
    END { print n+0 }'
)
```

The number of primer pairs with penalty ≤ 1 is printed (in feedback mode) and the primary output FASTA file is confirmed.

⟨*primers: report count and outputs*⟩≡
```
log "   Primer pairs with penalty <= 1: $count"
  log "   primers.fasta created"
```

The best-scoring primer pair is extracted as the first record in `primers.fasta`. Because the output is written as a two-sequence FASTA-like block (forward then reverse), the best pair is stored by taking the first 4 lines.

⟨*primers: save best primer pair*⟩≡
```
head -n 4 $safe_target/primers.fasta \
     > $safe_target/best_primers.fasta
    log "   Best scoring primer pair saved to best_primers.fasta"
```

6.1 In silico specificity testing

The best primer set is tested using `scop`. The database is provided via `-d $SCOPDB` and the target taxon id file is provided via `-t`. The resulting report is written to `scop.out`.

⟨*primers: run scop*⟩≡
```
log "   In silico testing best primer pair"
scop -d "$SCOPDB" \
     -t "$safe_target/taxids.txt" "$safe_target/best_primers.fasta" \
    > "$safe_target/scop.out"
```

6.2 Per-target completion

After all stages are complete (or skipped under check mode), a short completion message is printed for the current target.

⟨*pipeline: per-target completion log*⟩≡
```
log "Finished run for $safe_target
  "
```

7 Run completion and timing

The pipeline records an end timestamp in `logs/timer.log` and prints a short summary (in feedback mode). This provides a minimal wall-clock runtime record for the full run.

⟨*pipeline: end-of-run timer log*⟩≡
```
{
echo
echo "End:"
date
} >> logs/timer.log
log "$(tail -n 2 logs/timer.log)"
log "Pipeline complete."
```